



Validation de systèmes sur puce complexes du niveau transactionnel au niveau transfert de registres

Zeineb Belhadj Amor

► To cite this version:

Zeineb Belhadj Amor. Validation de systèmes sur puce complexes du niveau transactionnel au niveau transfert de registres. Micro et nanotechnologies/Microélectronique. Université de Grenoble, 2014. Français. NNT : 2014GRENT083 . tel-01228059

HAL Id: tel-01228059

<https://theses.hal.science/tel-01228059>

Submitted on 12 Nov 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Micro & Nano Électronique**

Arrêté ministériel : 7 août 2006

Présentée par

Zeineb BELHADJ AMOR

Thèse dirigée par **Dominique BORRIONE**
et codirigée par **Laurence PIERRE**

préparée au sein du laboratoire **TIMA**
et de l'École Doctorale **Électronique, Électrotechnique, Automatique
et Traitement du Signal**

Validation de systèmes sur puce complexes du niveau transactionnel au niveau transfert de registres.

Thèse soutenue publiquement le **17 Décembre 2014**,
devant le jury composé de :

Mr. Bruno ROUZEYRE

Professeur à l'Université Montpellier 2, Rapporteur

Mr. Philippe COUSSY

Professeur à l'Université de Bretagne Sud, Rapporteur

Mr. Jean-Louis LANET

Professeur à l'Université de Limoges, Président

Mme. Laurence PIERRE

Professeur à l'Université Joseph Fourier, Co-Directeur de thèse

Mme. Dominique BORRIONE

Professeur à l'Université Joseph Fourier, Directrice de thèse



à mon adorable petit ange Mayar ...

Remerciements

Je tiens tout d’abord à remercier les personnes qui ont permis la soutenance de cette thèse :

- Philippe COUSSY et Bruno ROUZEYRE qui m’ont fait l’honneur d’être les rapporteurs de thèse malgré leur charge de travail.
- Jean-Louis LANET qui a accepté de présider ce jury.
- Dominique BORRIONE qui a dirigé cette thèse pour sa patience, ses remarques pertinentes et ses critiques constructives.
- Laurence PIERRE qui m’a encadré, conseillé et aidé. Je la remercie grandement de m’avoir guidé tout au long de cette expérience. Je tiens aussi à la remercier également pour son investissement courant, sa disponibilité, sa franchise, et sa pédagogie. Merci aussi de la confiance durant toutes ces années.
- Je remercie les membres du laboratoire TIMA, et particulièrement de l’équipe VDS dans laquelle j’ai été accueillie durant ma thèse.
- Je remercie les anciens doctorants, les doctorants, et les non doctorants du laboratoire TIMA : Yann, Amr, Lucas, Renaud et Alex qui étaient les premiers à m’accueillir dans l’équipe VDS. Laila pour son amitié et sa bienveillance, Négune pour sa gentillesse inestimable. Kévin pour les gourmandises matinales. Martial pour ses encouragements. Lydie pour son énergie communicative. Sofiane pour son amitié, sa gentillesse, sa simplicité et tous les cafés qu’il m’a offert. Julien, Alice, Laurence, Ahmed, Frédéric pour nos nombreuses discussions autour des tables du RU. Héla, Chadi et Kais, Brice, Rshdee qui ne manquent pas de passer me voir et de demander de mes nouvelles. Chacun à votre façon, vous m’avez appris des choses dont je me souviendrai.

Je remercie l’équipe pédagogique de le Phelma pour leur accueil chaleureux. Un merci particulier à Katell Morin-Alloy, Régis Leveugle, François Portet, François Cayre, Laurent Fesquet, Lorena Anghel, Michèle Devignes, Michele Portolan et Mounir Benabdenbi pour m’avoir permis de participer à l’enseignement de leurs étudiants.

Un grand merci à tous les membres de la famille BEN ISMAIL, qui m'ont reçu comme un membre de leur famille, pour leur gentillesse, leur soutien, leur encouragements. Qu'ils trouvent ici l'expression de ma gratitude la plus profonde.

Je souhaite remercier mes amies et mes proches pour toutes leurs attentions. Mes remerciements s'adressent particulièrement à mon amie Islem pour sa présence et son aide aux moments difficiles.

Une profonde gratitude à mon mari Kamel, pour sa patience, sa générosité, son soutien sans failles, son amour et sa confiance, à mes soeurs : Nada, Hajer, Chiraz et sa princesse Line, mon cher frère Ahmed, pour leur inconditionnel et inestimable amour. Ma reconnaissance s'adresse particulièrement à mes parents Ali et Aïda qui m'ont continuellement soutenu comme seulement un père et une mère savent faire. C'est à vous tous que je dédie ce travail.

Enfin, un merci particulier à une personne qui n'a cessé d'être à mes côtés depuis le début de son existence, qui à chaque jour me donne un bonheur et une joie immenses, mon adorable petit ange Mayar.

J'en oublie certainement encore et je m'en excuse.
Encore un grand merci à tous pour m'avoir conduit à ce jour mémorable.

Table des matières

1	Introduction	1
	Préambule	1
1.1	Les limitations des flots de conception classiques	2
1.2	Un nouveau niveau d'abstraction : ESL (<i>Electronic System Level</i>)	4
1.2.1	Les critères de classement des niveaux d'abstraction	5
1.2.2	Vers un nouveau flot de conception	6
1.2.3	Les méthodes et les langages émergents avec la méthodologie ESL	9
1.2.3.1	La méthodologie TLM (<i>Transaction Level Modeling</i>)	9
1.2.3.2	Les langages émergents avec ESL	10
1.2.4	La vérification dans le flot ESL	12
1.2.4.1	Les techniques de vérification	12
1.2.4.2	La vérification des modèles SystemC	15
1.3	Motivations et contributions	15
1.3.1	Le raffinement des assertions TLM	16
1.4	Organisation du document	18
2	Langages de description et niveaux d'abstraction	21
	Introduction	22
2.1	Les niveaux d'abstraction du flot RTL	22
2.2	Les niveaux d'abstraction dans le flot ESL	26
2.2.1	Les langages de modélisation de haut niveau	26
2.2.1.1	La bibliothèque SystemC	26
2.2.1.2	La modélisation avec la bibliothèque SystemC-TLM	32
2.2.1.3	Exemple illustratif : le pv_dma	37
2.2.2	Les niveaux d'abstraction	43
2.2.2.1	Les niveaux d'abstraction Transactionnels	44
2.2.2.2	Le flot ESL détaillé	46
2.3	Le raffinement des modèles	48

2.4	Bilan	49
3	Contexte : La vérification à base d'assertions	51
	Introduction	52
3.1	La spécification à base d'assertions :PSL	52
3.1.1	Les opérateurs FL	54
3.1.1.1	Le sous ensemble simple PSL_{ss} (<i>Simple Subset</i>)	56
3.1.1.2	Les opérateurs faibles et forts	57
3.1.1.3	Les niveaux de satisfaction des FL	57
3.1.2	La sémantique de PSL	58
3.1.2.1	La sémantique des opérateurs temporels FL	59
3.1.2.2	La sémantique des SERES	62
3.2	La vérification dans les flots de conception	64
3.2.1	La vérification au niveau RTL : Horus	65
3.2.2	La vérification au niveau TLM	67
3.2.2.1	Les approches au niveau <i>Cycle Accurate</i>	67
3.2.2.2	Les approches au niveau transactionnel	69
3.2.2.3	L'outil ISIS	75
3.2.2.4	L'utilisation des SERES et de l'opérateur next	81
3.3	Bilan	83
4	Le raffinement des assertions	85
	Introduction	86
4.1	Les motivations et enjeux du raffinement	86
4.1.1	Les différences structurelles	86
4.1.2	Les différences temporelles	87
4.1.3	L'exemple du pv_dma	90
4.2	Travaux sur le raffinement des assertions	91
4.2.1	Approches à base de transacteurs	91
4.2.2	Approche de Ecker et al.	92
4.2.2.1	La transformation des assertions	93
4.2.2.2	Les restrictions	94
4.3	Solution de raffinement	95
4.3.1	Premières règles de raffinement	95
4.3.1.1	Règles de transformation impliquant l'opérateur <code>before</code>	95
4.3.1.2	Règles de transformation impliquant l'opérateur <code>until</code>	101
4.3.1.3	Règles de transformation impliquant l'opérateur d'implication logique	102
4.3.2	Alternative :utilisation des SERES	104

4.4	Bilan	105
5	Mise en oeuvre	107
	Introduction	108
5.1	Outil de raffinement des propriétés PSL	109
5.1.1	Les étapes de pré-raffinement	111
5.1.2	Le processus de raffinement	111
5.1.3	Réutilisation des fichiers Excel	121
5.2	Extensions d'ISIS	122
5.2.1	Prise en compte des fronts montants ou descendants de signaux autres que des signaux de type horloge	122
5.2.2	Possibilité de consulter les valeurs des signaux	122
5.3	L'implémentation des SERES	124
5.3.1	Définition des moniteurs SERE élémentaires	124
5.3.1.1	Moniteurs SERE dans Horus	124
5.3.1.2	Moniteurs SERE dans ISIS	126
5.3.2	Méthode d'interconnexion des SERES	132
5.3.2.1	Interconnexion des moniteurs FL élémentaires	132
5.3.2.2	Interconnexion des moniteurs SERE élémentaires	132
	Bilan	134
6	Expérimentations	135
	Introduction	136
6.1	Cas d'étude : Plateforme de décodage vidéo Motion—JPEG	136
6.1.1	Plateforme et assertions transactionnelles	136
6.1.1.1	P_1 : Intégrité des données	137
6.1.1.2	P_2 : Non perte des données	139
6.1.2	Plateforme et assertions RTL	140
6.1.2.1	Norme Wishbone	140
6.1.2.2	Raffinement des assertions	141
6.2	Cas d'étude : Plateforme de compression d'images	144
6.2.1	Plateforme et assertions transactionnelles	144
6.2.1.1	P_3 : Configuration du DMA_a	145
6.2.1.2	P_4 : Configuration de la FFT	147
6.2.1.3	P_5 : Non perte des paquets en entrée	148
6.2.2	Plateforme et assertions raffinées	149
6.2.2.1	La norme AMBA-AHB	149
6.2.2.2	Le raffinement des assertions	150
6.2.2.3	Exemple de raffinement en utilisant les SERES	159

6.3	Résultats et analyses	161
6.3.1	Complexification des assertions raffinées	161
6.3.2	Analyse des performances	162
6.3.2.1	Plateforme de décodage vidéo (MJPEG)	163
6.3.2.2	Plateforme de compression d'images	164
6.3.3	Synthèse	169
6.4	Bilan	169
7	Conclusion et perspectives	171
7.1	Contributions	171
7.2	Travaux futurs	173
7.2.1	Travaux à court terme : génération des chronogrammes	173
7.2.2	Perspectives	173
A	Les preuves de la transformation formelle des règles	175
A.1	Preuve de la règle de transformation T_3	175
A.2	Preuve de la règle de transformation T_4	175
A.3	Preuve de la règle de transformation T_5	176
A.4	Preuve de la règle de transformation T_6	176
B	Détails sur le Flot de raffinement	177
B.1	Modèle du fichier XML des règles de raffinement	177
B.2	Illustration détaillée du flot de raffinement	179
C	Résultats Expérimentations : Plateforme de décodage vidéo	181
C.1	Plateforme transactionnelle	181
C.1.1	Fichier d'instrumentation	181
C.1.2	Extraits des traces de simulation	182
C.2	Plateforme Raffinée	185
C.2.1	Étapes de raffinement	185
C.2.2	Fichier d'instrumentation généré	186
C.2.3	Extrait des traces de simulation	187
D	Résultats Expérimentations : Plateforme de traitement d'images	193
D.1	Plateforme transactionnelle	193
D.1.1	Propriété P_3	193
D.1.2	Propriété P_4	195
D.1.3	Propriété P_5	196
D.2	Raffinement selon le protocole Wishbone	198
D.2.1	Propriété P_3 raffinée	198

D.2.2	Propriété P_4 raffinée	200
D.2.3	Propriété P_5 raffinée	203
D.3	Raffinement selon le protocole AHB	207
D.3.1	Propriété P_3 raffinée	207
D.3.2	Propriété P_4 raffinée	213
D.3.2.1	Raffinement en FL	214
D.3.2.2	Raffinement en SERE	218
D.3.3	Propriété P_5 raffinée	222
E	Couche modélisation et transferts en rafale	225
	Bibliographie	227
	Glossaire	237
	Publications de l'Auteur	243

Table des figures

1.1	Flot de conception RTL d'un SoC (source : [GCMC ⁺ 05]-modifié)	3
1.2	Le manque à gagner en fonction de l'évolution de la complexité des SoCs.(source : [Bai04])	4
1.3	Le flot de conception idéalisé basé sur ESL.(source : [DR10] [Ger10])	7
1.4	Le flot de conception basé sur la plateforme.(source : [BM10])	9
1.5	Le temps moyen consommé par chaque tâche de vérification(source : [Fos11]).	16
1.6	Flot de vérification parallèle	17
1.7	La disparité des domaines temporels entre TLM et RTL	17
2.1	Le flot de conception partant du modèle RTL	23
2.2	La topologie crossbar des bus de communication (source :[ope10])	24
2.3	La représentation de l'architecture du bus wishbone	24
2.4	Différence entre un module hiérarchique et un module simple	28
2.5	conception basée sur les interfaces (source :[RSV97])	29
2.6	<i>Interface Method Call</i> et raffinement du support de communication	30
2.7	Le mécanisme de synchronisation dans SystemC	32
2.8	La structure de bibliothèque TLM-2(source :[OSC09])	33
2.9	Le schéma d'interconnexion dans TLM-2 (source :[Ayn09])	34
2.10	le déroulement de l' <i>Interface Method Call</i> dans TLM-2	35
2.11	Les phases des transports bloquants et non bloquants	37
2.12	Les modèles d'abstraction SystemC du DMA	39
2.13	Le diagramme de séquence des appels de fonctions pour faire une lecture en mémoire par le DMA	43
2.14	Les cas d'utilisation, les styles de codage et les mécanismes de programmation (source :[OSC09])	44
2.15	Les niveaux d'abstraction et cas d'utilisation (source :[Sch09])	46
2.16	Les modèles d'abstraction en SystemC-TLM	47
2.17	L'outil HLS GAUT (source :[Bou07])	47

2.18	La vérification du DUT par rapport au modèle de référence[RN03]	49
2.19	Un transacteur TLM-2 vers wishbone	49
3.1	Un exemple d'une <i>vunit</i> PSL	54
3.2	Exemples de traces pour la propriété P_1	55
3.3	Exemples de traces pour la propriété P_2	56
3.4	Exemples de traces pour la propriété P_3	57
3.5	Exemples de traces pour la propriété P_4	60
3.6	Exemples de trace qui satisfont la propriété P_5	61
3.7	Exemples de trace qui satisfait la propriété P_6	62
3.8	Exemple de trace pour la propriété P_7	65
3.9	Moniteur RTL généré par Horus pour la propriété P_8	67
3.10	Le flot de la génération des moniteurs proposé dans [Lah06]	70
3.11	L'architecture du langage UAL	71
3.12	Exemple d'implémentation d'un moniteur UAL [Ese08]	72
3.13	Le flot de la méthodologie de vérification proposée dans [GLD10]	74
3.14	Le flot pour la méthodologie de vérification proposée dans [Fer11]	75
3.15	L'implémentation du paradigme <i>Observable-Observer</i> dans ISIS[Fer11]	77
3.16	Moniteur global généré par ISIS pour P_8	77
3.17	Extrait de trace d'exécution du système RTL	82
3.18	Scénario d'exécution du transfert d'un bloc de données	82
4.1	Raffinement des interfaces des composants maître et esclave TLM	87
4.2	Exemples d'appels à des fonctions bloquantes et non bloquantes	88
4.3	Les chronogrammes d'une écriture simple pour le bus wishbone (source:[ope10]) et AHB (source:[ARM08])	89
4.4	Une opération d'écriture <i>burst</i> du bus Wishbone (source:[ope10])	89
4.5	Le raffinement temporel d'une opération d'écriture simple selon le bus AHB	90
4.6	Le raffinement temporel d'une opération d'écriture <i>burst</i> selon le bus AHB	91
4.7	Méthode de réutilisation des assertions transactionnelles (source:[BFP07])	91
4.8	Processus de transformation selon Ecker et al. (source:[Ste12])	93
4.9	Comportement temporel décrit par la contrainte temporelle de T_2	100
4.10	La complexification structurelle due au raffinement temporel de l'assertion P_9	104
5.1	Méthodologie de vérification proposée	108
5.2	La composition de l'outil	110
5.3	Interface de sélection	112
5.4	Exemple de fichier Excel généré	114
5.5	Exemple de fichier Excel de transformation des variables	115

5.6	Exemple d'un chronogramme Excel	115
5.7	Processus de la transformation temporelle	117
5.8	Arbre syntaxique de l'application de T_6 dans P_9	119
5.9	Moniteur Horus de P_1 [MAGB07]	125
5.10	Connecteur [*] monochrome	126
5.11	Structure des connecteurs [*] Horus(à gauche) et ISIS(à droite)	129
5.12	Trace de la propriété P_2	131
5.13	Interface du gestionnaire des jetons	131
5.14	Moniteur global généré par ISIS pour la formule FL P_8	133
5.15	Moniteur SERE global généré par ISIS	133
6.1	Plateforme MJPEG (source [GGP08])	137
6.2	Interfaces <i>driver</i> , <i>responder</i> du Wishbone	141
6.3	Protocole "poignée de main" du Wishbone (source :[ope10])	141
6.4	Chronogramme de la méthode write du processeur	142
6.5	Plateforme de compression d'images AstriumV2	145
6.6	Interface d'un maître AHB	150
6.7	Chronogramme correspondant à la fonction write_block et à ses paramètres	152
6.8	fichier Excel de transformation des variables globales de P_3	153
6.9	La fin d'un transfert en rafale AHB[ARM08]	156
6.10	Chronogramme correspondant à read_block_END (transfert en rafale) . .	157
6.11	M-JPEG : caractéristiques des propriétés TLM P_1 , et P_2	162
6.12	M-JPEG : complexification des propriétés raffinées P_1 , et P_2	162
6.13	Traitement d'images : caractéristiques des propriétés TLM P_3 , P_4 et P_5 . .	162
6.14	Traitement d'images : complexification des propriétés raffinées P_3 , P_4 et P_5	163
6.15	Plateforme de décodage vidéo au niveau transactionnel : temps CPU pour les propriétés P_1 et P_2	163
6.16	Plateforme de décodage vidéo au niveau transactionnel : nombre d'activa- tions des moniteurs de P_1 et P_2	164
6.17	Plateforme de décodage vidéo au niveau RTL : temps CPU pour les pro- priétés P_1 et P_2 raffinées	165
6.18	Plateforme de décodage vidéo au niveau RTL : nombre d'activations des moniteurs de P_1 et P_2 raffinées	165
6.19	Plateforme de traitement d'images au niveau transactionnel : temps CPU pour les propriétés P_3 , P_4 et P_5	166
6.20	Plateforme de traitement d'images au niveau transactionnel : nombre d'ac- tivations des moniteurs de P_3 , P_4 et P_5	166
6.21	Plateforme de traitement d'images hybride implémentant le protocole Wish- bone : temps CPU pour les propriétés P_3 , P_4 et P_5 raffinées	167

6.22	Plateforme de traitement d'images hybride implémentant le protocole Wishbone : nombre d'activations des moniteurs de P_3 , P_4 et P_5 raffinées	167
6.23	Plateforme de traitement d'images hybride implémentant le protocole AHB : temps CPU pour les propriétés P_3 , P_4 et P_5 raffinées	168
6.24	Plateforme de traitement d'images hybride implémentant le protocole AHB : nombre d'activations des moniteurs de P_3 , P_4 et P_5 raffinées	169
B.1	Flot complet détaillé	180
C.1	Interface de sélection des méthodes à transformer	185
C.2	Correspondance entre les signaux et la fonction write et ses paramètres . .	186
C.3	Extrait la trace d'exécution en utilisant Gtkwave	188
D.1	Correspondance entre la fonction write_block et ses paramètres et les signaux AHB	207
D.2	Chronogramme correspondant à read_block_END (transfert simple) . . .	213

Introduction

*“It’s a very humbling experience to make a multi-million-dollar mistake,
but it is also very memorable.”*

Dr.Fredrick Brooks- No Silver Bullet, Information Processing-1986

Préambule

Un Système sur puce ou SoC (*System-on-Chip*) présent dans les systèmes embarqués est un ensemble de composants matériels et de logiciels conçus et intégrés dans une seule puce électronique pour réaliser les fonctionnalités demandées. Typiquement, un SoC est composé de périphériques d’entrée/sortie, d’éléments de calcul et de traitement (microprocesseur, DSP¹), d’unités de sauvegarde des données (mémoire) communiquant via une structure d’interconnexion (bus, NoC²). L’ensemble est inclus dans un produit destiné à des utilisateurs potentiels. Les systèmes embarqués alimentent les innovations faites dans divers domaines : la robotique, l’aviation, les télécommunications, l’automobile, la sécurité, la médecine, etc. et leur criticité dépend du champ d’application. Depuis la création des premiers circuits intégrés en 1958, la complexité de ces circuits n’a pas cessé de s’accroître. Aujourd’hui, par exemple, on ne parle plus de téléphones portables mais de téléphones intelligents (*Smartphone*). Hormis sa fonction première qui est de téléphoner, ce dernier est devenu suffisamment sophistiqué pour servir aussi de GPS, d’appareil photo, de télévision, de lecteur de musique, de console de jeux, etc. et inclure une multitude d’applications. La complexité d’un SoC se mesure en nombre de transistors. En 1965, Gordon Moore, futur cofondateur de l’entreprise Intel, avait estimé que le nombre de transistors présents sur une puce électronique allait doubler à prix constant tous les ans. En se basant sur des constats empiriques, il rectifia par la suite son estimation en portant à dix-huit mois le rythme de doublement. Cette estimation nommée par la suite la “loi

¹Digital Signal Processor

²Network on Chip

de Moore” s’est effectivement vérifiée et décrit parfaitement l’évolution des SoCs. Les premiers microprocesseurs Intel créés en 1971 contenaient quelques milliers de transistors, actuellement, on parle de milliards de transistors [Cor12].

Les architectures existantes permettent à une même plateforme³ matérielle de réaliser plusieurs applications, présentant ainsi une grande flexibilité fonctionnelle au prix d’une complexité supplémentaire. Cette évolution spectaculaire a plusieurs répercussions sur les processus de conception et de fabrication de ces systèmes. En effet, dans le cadre d’une production massive, les entreprises sont amenées à concevoir des systèmes de plus en plus complexes et performants dans un TTM (*Time-To-Market*) de plus en plus réduit et à des prix compétitifs. La réussite d’un tel défi repose sur l’utilisation d’une méthodologie de conception efficace et rigoureuse adaptée aux spécificités de ces systèmes. Ainsi, on assiste à une véritable révolution dans le monde de la conception et de la vérification des micro-architectures.

Dans cette introduction, nous allons présenter brièvement le contexte général de cette thèse en passant par une étude de l’évolution des processus de conception et de vérification des SoCs. A l’issue de cette étude, la problématique traitée sera identifiée et présentée. Finalement, un aperçu du contenu des prochains chapitres permettra de comprendre la structuration de ce manuscrit.

1.1 Les limitations des flots de conception classiques

Une méthodologie de conception est un ensemble ordonné de méthodes et de techniques mises en place pour gérer la complexité des SoCs et qui constituent un flot de conception. Les premiers flots de conception sont apparus avec le passage du niveau portes logiques au niveau transfert des registres (RTL⁴) principalement grâce aux langages de description du matériel (*HDL : Hardware Description Language*) et aux outils de synthèse.

En se référant à l’histoire d’évolution des SoCs, on peut distinguer deux principaux flots de conception. Dans un premier temps, on parle d’un flot de conception classique tel que décrit par la Figure 1.1. Le processus commence par le partitionnement des fonctionnalités demandées en composants matériels (HW⁵) et logiciels (SW⁶). Ensuite deux branches indépendantes se distinguent selon le type du composant à développer. La conception des modules matériels commence par une description en HDL, écrite en VHDL[VHD02] ou en Verilog[VER01]. Une fois simulée et synthétisée, cette description permettra la mise au point d’un premier prototype du système. De l’autre côté, une autre équipe

³le terme plateforme peut avoir plusieurs significations. Dans le cadre de cette thèse on l’utilise pour désigner l’infrastructure sur laquelle les applications logicielles sont déployées. C’est le standard selon lequel le système va être créé.

⁴Register Transfer Level

⁵Hardware

⁶Software

démarre le développement du logiciel indépendamment de l'architecture et des considérations matérielles. Selon cette approche la vérification du logiciel n'est possible que lorsque le prototype est prêt. C'est souvent la première opportunité pour la vérification et la validation du logiciel. De plus, si un problème matériel est détecté après la réalisation du prototype, tout le processus de conception du matériel doit être repris, impliquant des ressources coûteuses. Entre temps, le développement du logiciel sera bloqué en attente de la prochaine version du matériel. Plusieurs itérations sont nécessaires avant que le système final soit prêt.

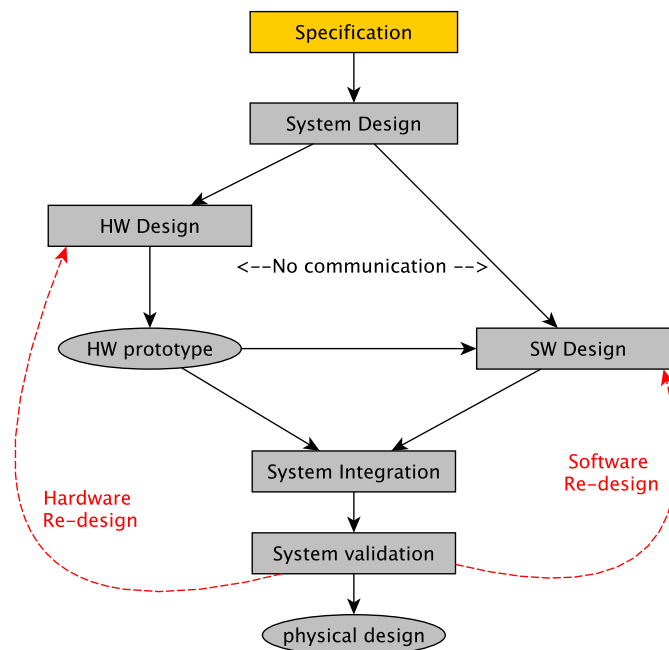


FIGURE 1.1 – Flot de conception RTL d'un SoC (source : [GCMC⁺05]-modifié)

Les limites de cette méthodologie de conception centrée sur le matériel ont commencé à se faire sentir depuis près de 20 ans avec l'évolution des techniques de miniaturisation. Désormais, un SoC ou MPSoC (*Multi-Processor System-on-Chip*) peut contenir plusieurs microprocesseurs et DSPs et une multitude de composants hétérogènes (analogiques, numériques) et de programmes élaborés. Le décalage entre les deux branches du processus de conception se traduit par des pertes de productivité et une augmentation des coûts de plus en plus conséquents. Depuis 1990, l'apparition des IPs (*silicon intellectual property*) et par conséquent du concept de réutilisation d'un côté, et des techniques de "co-design" et de "co-verification" de l'autre, ont permis d'économiser un temps considérable et de gagner en efficacité. Toutefois elles n'ont pas réussi à combler l'écart entre le développement du logiciel et du matériel. Selon [Kea11][Fos09][Bai04], la vérification est le principal goulot d'étranglement du flot et pour la plupart des concepteurs, elle con-

somme entre 50% et 80% de l'activité de conception pour plusieurs raisons : contrairement aux prouesses réalisés dans les techniques d'intégration (VLSI⁷) et les progrès effectués grâce aux outils de conception et de synthèse des modèles RTL, les techniques de vérification dans les plus bas niveaux de description n'ont pas suivi la même évolution. Le nombre exponentiel de composants dans le SoC augmente la probabilité d'erreurs et les rend difficilement repérables. De plus, la vérification du code HDL, étant très lente, limite les capacités de vérification et creuse un écart important par rapport à la capacité de conception. La Figure 1.2 qui date d'une dizaine d'années, témoignait déjà du décalage entre la conception, la vérification et la fabrication. Au final, ce processus de conception engendre un manque à gagner dû à la découverte tardive des défauts et à l'inefficacité de la vérification fonctionnelle au niveau RTL pour les SoCs complexes. Le compromis coût/qualité se traduit par un besoin impératif de nouvelles méthodologies de conception permettant de mieux gérer la complexité des SoCs en relevant encore une fois le niveau d'abstraction, afin de résoudre les problèmes de vérification.

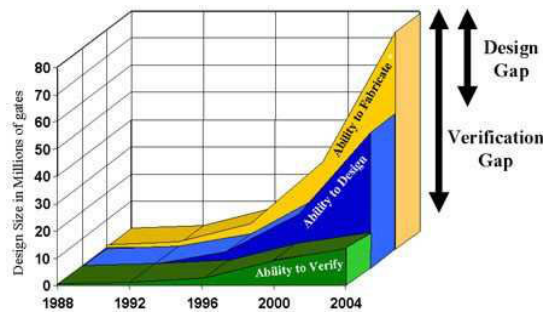


FIGURE 1.2 – Le manque à gagner en fonction de l'évolution de la complexité des SoCs.(source : [Bai04])

1.2 Un nouveau niveau d'abstraction : ESL (*Electronic System Level*)

L'abstraction vise à diminuer l'effort nécessaire pour spécifier les fonctionnalités demandées en omettant volontairement les détails inutiles dans les niveaux élevés de la conception [DR10]. Il n'existe pas une définition exacte du terme ESL. Globalement, c'est le niveau d'abstraction qui doit commencer en dessus de RTL. Les auteurs de [MBP07] le définissent comme étant "l'utilisation d'un niveau d'abstraction approprié afin d'améliorer la compréhension du système et l'implémentation efficace des fonctionnalités à moindre coût". En pratique, il est associé à un ensemble de méthodologies complémentaires de conception et de vérification permettant de procéder par raffinements successifs pour passer

⁷Very Large Scale Integration

de la spécification à l'implémentation. Ainsi, il n'existe pas une seule variante du flot de conception ESL. Les différentes variantes utilisées dans l'industrie tournent autour de trois principes clés :

- la spécification fonctionnelle dirigée par les exigences,
- l'exploration des architectures et l'évaluation des performances,
- le raffinement incrémental des spécifications fonctionnelles en descriptions architecturales.

La conception fonctionnelle permet de décrire le comportement du système d'un point de vue interne incluant la structure, les communications et les activités élémentaires. Elle est centrée sur l'application et ne prend pas en compte les considérations physiques et technologiques et les langages de programmation. La conception architecturale, quant à elle, est centrée sur la création d'une plateforme matérielle adéquate pour le déploiement de l'application.

Les différentes étapes de raffinement, les niveaux d'abstraction considérés et les outils utilisés permettent de différencier un flot de conception ESL d'un autre.

1.2.1 Les critères de classement des niveaux d'abstraction

Les auteurs de [GLMS02] proposent un ensemble de critères indépendants selon lesquels on peut mesurer la précision d'un modèle par rapport à son implémentation. Un modèle ne représente généralement pas le SoC au même niveau de granularité que son implémentation. Il est donc plus simple, permettant à la fois de concevoir le SoC en plusieurs étapes et de simuler son exécution. C'est la mesure de cette simplification qui permet de définir les niveaux de précision selon différents critères :

- la précision structurelle (*Structural Accuracy*) : Ce critère mesure le degré de fidélité de la structure du modèle par rapport à celle de l'implémentation. Ceci inclut l'identification des composants et leur partitionnement en modules matériels ou logiciels. Pour un module matériel, la précision structurelle concerne la description de sa composition interne et de son interface. Pour un composant logiciel, la précision concerne l'identification des différentes tâches et de leur correspondance par rapport au jeu d'instructions du processeur à l'implémentation.
- la précision temporelle (*Timing accuracy*) : Ce critère consiste à mesurer le degré de précision temporelle du modèle par rapport à son implémentation. Il est déterminé selon la présence ou l'absence d'annotations temporelles permettant de simuler les latences de transmission, les délais de traitement et de communication et le mode de synchronisation (à base d'horloge ou d'appels de fonctions). Quand le modèle ne contient aucune notion de temps, il est dit "non-temporisé" (*untimed*). Le niveau de granularité maximal de temps est atteint quand les valeurs temporelles sont exprimées en termes de cycles d'horloges. Dans ce cas, on parle du niveau *cycle-*

accurate.

- la précision fonctionnelle (*Functional accuracy*) : Ce critère se place dans une vision fonctionnelle du modèle et s'intéresse à évaluer la fidélité des fonctionnalités du modèle par rapport à son implémentation (partielle ou totale). Dans les modèles abstraits, les fonctionnalités complexes peuvent être volontairement omises.
- la précision de l'organisation des données (*Data organisation accuracy*) : Ce niveau concerne la mesure de la précision des types de données, et donc des interfaces par rapport à celles de l'implémentation. Dans les modèles au niveau système par exemple, les données peuvent être représentées par des valeurs entières. La précision est maximale quand les données sont décrites par des vecteurs de bits (*bit-accurate*).
- la précision du protocole de communication (*Communication protocol accuracy*) : C'est une mesure du niveau de précision du support des communications par rapport aux protocoles adoptés dans l'implémentation. Ce critère prend en considération le niveau de précision structurel, temporel dans la description des communications.

Il faut noter que ces critères ne permettent pas nécessairement d'évaluer la précision d'un modèle dans sa globalité. Dans le cadre d'une description modulaire, les mesures peuvent intéresser un module ou même un sous module.

1.2.2 Vers un nouveau flot de conception

Suite à l'étude de plusieurs flots de conceptions utilisés [KB06] [MBP07] [Fuj03] [HFK⁺07], Drechsler et Rogin[DR10] ont proposé une représentation idéalisée d'un flot de conception ESL composé de six étapes. Selon ce flot illustré par la Figure 1.3, le concepteur commence par transformer le document de spécification textuelle en spécification exécutable. Ce modèle est l'équivalent du modèle non-temporisé présenté dans la section précédente. Grâce à ce modèle, l'exploration d'architecture est possible en prenant en compte les contraintes de performances, temps, consommation, espace, complexité et *Time-To-Market*. Ensuite, le processus de partitionnement désignera les fonctionnalités qui seront implémentées en matériel ou en logiciel. Cette étape commence par la production d'un modèle logiciel de la plateforme matérielle au niveau système qui servira de prototype virtuel pour le développement logiciel d'un côté, et de modèle de référence pour les modèles matériels moins abstraits de l'autre côté. Ce modèle sera raffiné successivement avant de passer en entrée des outils de synthèse de haut niveau. Les étapes de conception et d'implémentation des composants matériels donneront en sortie les modèles RTL synthétisables. Finalement, le résultat de l'intégration des composants matériels et logiciels amènera à la production du système complet.

La présence d'une plateforme virtuelle simplifiée, rapide à simuler et fidèle aux exigences initiales est la base de ce flot descendant. Elle facilite la communication entre les deux équipes de développement matériel et logiciel et leur permet un développement

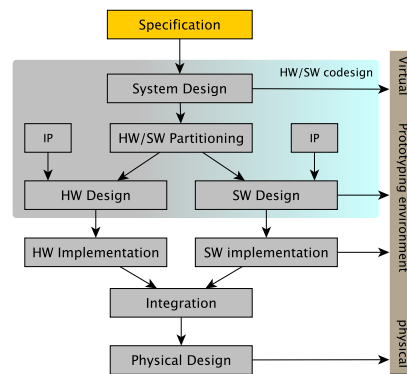


FIGURE 1.3 – Le flot de conception idéalisé basé sur ESL.(source : [DR10] [Ger10])

parallèle (*co-design*) en se référant à une base commune. Les plateformes virtuelles sont faciles à tester et à modifier et offrent de meilleures performances par rapport à l'implémentation. Ces plateformes peuvent être spécifiées à différents niveaux d'abstraction. Selon [BM10], il est possible de les classer en deux groupes en se basant sur leur niveau d'abstraction :

- niveau fonctionnel (*functionally accurate*) : à ce niveau la plateforme virtuelle est notamment utilisée pour le développement logiciel. On peut distinguer deux sous niveaux :
 - Application View : à ce niveau, on parle aussi d'une spécification exécutable car elle ne reflète que les fonctionnalités demandées indépendamment de l'implémentation. L'avantage de cette spécification est la rapidité de la simulation (de l'ordre de +150MIPS⁸[BM10]). Ces performances sont atteintes en exécutant l'application comme un processus natif dans une station de travail sans inclure aucune caractéristique de la plateforme cible telle que la description du jeu d'instructions du processeur.
 - Programmer's View (PV) : ce niveau est le résultat du raffinement de la spécification exécutable en injectant quelques détails de l'architecture matérielle tels que la structure du processeur, des registres, du jeu d'instructions, etc. Ce modèle est généralement basé sur un processeur non pipeliné permettant d'évaluer les fonctionnalités sur l'architecture en dépit d'une simulation moins rapide (entre 40MIPS et 60MIPS[BM10]).
- Cycle Approximative (CA) : ce niveau peut être vu comme le raffinement du niveau PV dans le cadre d'une démarche centrée sur l'application (*Application-based design*) ou encore comme le point de départ d'une démarche centrée sur la plateforme

⁸MIPS : Million Instructions Per Second.

(*platform-based design*) auquel cas, elle correspond à l'implémentation d'un prototype physique résultant du *mapping* des fonctionnalités sur la plateforme physique. Dans les deux cas, les détails temporels approximatifs sont inclus donnant une estimation des temps de calcul, des latences, des temps de communication. Cette plateforme supporte aussi l'exécution du système d'exploitation temps réel et de la couche d'abstraction matérielle (HAL⁹). Par conséquent, la simulation est plus lente (entre 1 et 10MIPs). En contrepartie, des mesures plus précises peuvent être réalisées pour l'exploration d'architecture et l'évaluation des performances.

Table 1.1 – Les outils de plateformes virtuelles des industriels(source :[BM10]-updated)

vendors	functionally accurate	cycle approximative
Synopsys	Virtualizer (Dev. kit)	Platform Architect
Mentor Graphics		vista
WindRiver	Simics	
ARM	Fast Models	
Imperas (OVP)	OVPsim	
Carbon	SoC Designer Plus	SoC designer Plus
Cadence	Virtual System Platform (VSP)	Rapid Prototyping Platform (RPP)

Bailey et Martin ont aussi proposé dans [BM10], en 2010, un tableau récapitulatif des principaux outils industriels utilisés pour le développement de plateformes virtuelles. Nous proposons ici, une mise à jour de ce tableau pour prendre en considération les fusions, les alliances et les évolutions dues au dynamisme du marché. On note par exemple, que depuis 2010, Synopsis a acquis VaST et Coware et Intel/Windriver a acquis Virtutech. Il est à noter aussi que les outils Cadence ont été proposés en 2012 et n'apparaissaient pas par conséquent dans l'ancien tableau. Comme le montre le tableau 1.1, la tendance actuelle des industriels est de suivre un flot de conception descendant orienté par l'application[BM10]. Cela est dû à la démarche incrémentale permettant de passer par plusieurs niveaux d'abstraction et de vérifier au fur et à mesure les aspects liés à chaque niveau. L'approche de conception basée sur la plateforme illustrée par la Figure 1.4 peut néanmoins être intégrée dans ce flot dans le but de mieux guider les choix de partitionnement (HW/SW). Ceci est particulièrement utile dans les domaines d'application critiques (aviation[LBC⁺10]) ou quand l'évaluation des performances dépend des données en entrée (tel est le cas dans les plateformes de traitement d'images par exemple).

Il faut noter aussi qu'en pratique, le processus de conception n'est pas parfaitement linéaire. De plus, le raffinement incrémental peut engendrer des modèles hybrides contenant des éléments à des niveaux d'abstraction différents. Le passage d'un niveau d'abstraction à un autre est dans la plupart des cas (semi-)automatique.

⁹Hardware Abstraction Layer

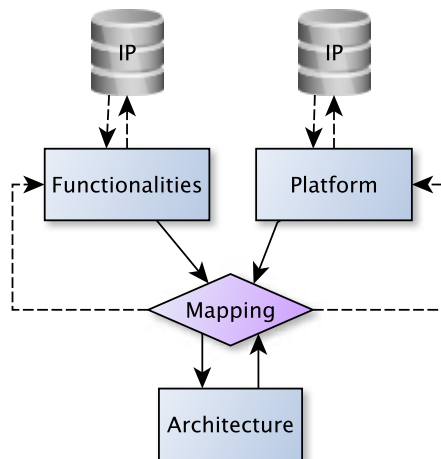


FIGURE 1.4 – Le flot de conception basé sur la plateforme.(source : [BM10])

Les travaux proposés dans cette thèse trouvent leur motivation dans cette approche de conception : les méthodes de vérification doivent suivre les étapes du flot de conception, via un raffinement de la spécification originale, en parallèle au raffinement de l'architecture.

1.2.3 Les méthodes et les langages émergents avec la méthodologie ESL

Travailler à un haut niveau d'abstraction a plusieurs avantages pour les équipes de conception et de vérification :

1. le modèle est plus simple et flexible,
2. les modifications sont plus faciles et moins coûteuses,
3. la vérification est plus rapide et efficace,
4. le temps de conception et de vérification est réduit.

ESL offre ces avantages en fournissant des méthodologies avancées basées sur une deuxième génération d'outils et de langages destinés à la conception, la synthèse et la vérification au niveau système (SL : System Level).

Dans cette section, nous donnons un bref aperçu des méthodologies et langages les plus importants impliqués dans la conception de haut niveau.

1.2.3.1 La méthodologie TLM (*Transaction Level Modeling*)

Cette méthodologie a été créée dans le cadre de la recherche d'un nouveau paradigme offrant une représentation intermédiaire reliant les spécifications au niveau RTL. Ainsi, elle constitue une partie intégrante de la méthodologie ESL. L'objectif de TLM est de

fournir un modèle de référence (*Golden Model*) aux différentes équipes de développement logiciel, matériel, d'analyse des architectures et de vérification. Ainsi, ce modèle doit apparaître très tôt dans le flot de conception, généralement, juste après l'étape de partitionnement.

Le principal objectif de TLM est de donner une représentation simplifiée des fonctionnalités complexes du système en faisant abstraction des détails matériels inutiles aux premières étapes du flot de conception. Il offre ainsi un moyen pour minimiser la quantité d'événements et d'informations à simuler. Par conséquent, un modèle transactionnel est jusqu'à 1000 fois plus rapide à simuler que son équivalent RTL[Ber05]. Cet atout majeur permet une vérification fonctionnelle plus étendue et nettement plus efficace par rapport aux limitations accrues de la vérification des modèles RTL.

Le concept clé de TLM est la séparation des aspects de communication et de calcul. Gajski a été parmi les premiers à inclure ce principe dans la conception du matériel[CG03]. Ce concept rejoint l'objectif de simplification de TLM en encapsulant les signaux de communication dans une structure particulière nommée "transaction" et permet par ailleurs une représentation modulaire et indépendante de l'architecture et du protocole de communication. En effet, dans une description TLM, les composants sont des modules contenant des processus concurrents qui exécutent leurs comportements associés. La communication entre ces modules est assurée par des canaux de communication transportant les transactions. Les interfaces TLM masquent le protocole de communication implémenté dans les canaux. Les modules accèdent aux canaux par l'intermédiaire de ports connectés aux interfaces. La flexibilité et la rapidité du modèle transactionnel le rend aussi idéal pour l'exploration de l'espace des architectures, l'évaluation des performances et favorise la réutilisation.

1.2.3.2 Les langages émergents avec ESL

Deux principaux langages de conception du matériel ont été développés conjointement à ESL : SystemC[SC005] et SystemVerilog[SV005]. Le premier est une extension du langage C++ sous forme d'une bibliothèque de classes et de macros qui enrichit le langage source par les structures nécessaires pour modéliser la concurrence, les communications et le temps. SystemC est né de l'initiative de OSCI¹⁰. Les efforts et la persévérance de ce groupe ont permis l'évolution de cette bibliothèque *open-source* en un standard IEEE¹¹ Std.1666-2005 rapidement adopté par les industriels. Lors de la migration vers un flot de conception basé sur ESL, SystemC s'est imposé comme étant un langage de conception

¹⁰ *Open SystemC Initiative*, une association à but non lucratif dédiée à définir et promouvoir SystemC (<http://www.systemc.org>).

¹¹ Institute of Electrical and Electronics Engineers, <http://www.ieee.org> et <http://standards.ieee.org/> est un organisme international qui définit entre autres des normes pour la conception et l'usage de systèmes électriques et électroniques.

de haut niveau incontournable. En effet, les langages C/C++ jouaient déjà un rôle majeur dans le développement des logiciels embarqués et l'adoption d'une solution similaire pour la conception du matériel s'avère un choix judicieux. Cette bibliothèque est bâtie sur le principe de la séparation des aspects de communication et de calcul afin de permettre plus de modularité et de flexibilité au modèle. Ce principe commun montre que le style de représentation de SystemC correspond parfaitement à celui de TLM.

La méthodologie TLM est indépendante des langages de conception du matériel (SystemC, SystemVerilog, SpecC, etc.). Les concepts supportés peuvent être implémentés par plusieurs langages. En effet, dans [Duc07], Laurent Ducouso, le responsable de la vérification à la division de STMicroelectronics à Grenoble, a indiqué que STMicroelectronics a commencé à implémenter des modèles transactionnels en utilisant des processus C et Unix depuis 2000. Ainsi, SystemC n'est pas indispensable pour TLM. Cependant, depuis 2005, la standardisation de ce dernier a largement encouragé son adoption par les concepteurs. De plus, l'implémentation d'une bibliothèque TLM[OSC09] en amont de ce langage a fait de SystemC un standard idéal pour la modélisation du matériel réutilisable, interopérable au niveau transactionnel. L'alliance SystemC-TLM est parfaite : les concepteurs commencent par identifier les composants matériels et logiciels du système dont chacun donnera lieu à un module SystemC. La bibliothèque TLM est ensuite utilisée pour modéliser la communication entre ces modules. Ce lien est tellement fort que l'OSCI a mis en place un groupe de travail chargé de définir le standard Accellera¹² SystemC-TLM. Cette alliance offre une panoplie de niveaux d'abstraction partant du niveau système au niveau RTL.

Il y a deux aspects dans lesquels on peut mesurer les avantages de SystemC-TLM par rapport à RTL : la rapidité et la qualité. Actuellement, la vitesse de simulation d'un modèle SystemC-TLM est 10,000 fois plus rapide que celle au niveau RTL. La simplicité du modèle SystemC ajoutée à la rapidité d'exécution engendrent systématiquement une amélioration de l'efficacité de la simulation et du débogage.

De l'autre côté SystemVerilog est un langage de description, de vérification du matériel orienté objet basé sur le standard Verilog Std.IEEE-1364. L'avantage de ce langage est qu'il offre un niveau d'abstraction supérieur à Verilog (donc au niveau RTL) en conservant la même syntaxe. Ainsi, il est facilement adopté par les concepteurs du matériel utilisant Verilog et offre une simulation plus rapide que le modèle RTL. Cependant, ce niveau d'abstraction n'est pas suffisamment élevé pour la spécification des micro-architectures ce qui le rend difficilement accessible aux concepteurs du logiciel. Selon G.Martin (Cadence Berkeley Labs)[Mar03], " SystemC n'est pas un langage HDL optimal et SystemVerilog n'est pas un bon langage pour la modélisation au niveau système". Il est possible de

¹²Accellera : *acceleration en italien*, une organisation de standardisation indépendante à but non lucratif qui travaille en collaboration avec IEEE. Les standards sont souvent adoptés à maturité comme standards IEEE (<http://www.accellera.org>).

combiner les deux langages dans un seul flot de conception interagissant par l'intermédiaire de la couche d'abstraction TLM. Le modèle résultant permet aux concepteurs d'effectuer la co-simulation.

1.2.4 La vérification dans le flot ESL

Selon le standard IEEE Std 1012-2004, le terme “vérification” désigne “la démonstration objective de la cohérence, de la complétude et de la correction d'un produit tout au long de son cycle de développement et entre tous les stades de ce cycle”[VV005]. En d'autres termes, la vérification consiste à s'assurer que le système a été construit correctement. La vérification fonctionnelle vise à s'assurer que le système effectue correctement les tâches conformément à la spécification [WGR05]. Concrètement, l'objectif est d'identifier les comportements incorrects manifestés par le système et de s'assurer que l'implémentation est conforme à la spécification.

L'analyse des coûts des SoCs complexes[Avi10] a démontré qu'on ne peut pas viser une amélioration de la productivité sans passer par une révision du processus de vérification. L'un des principaux objectifs d'ESL est d'assurer la correspondance entre le modèle et les exigences et de détecter les erreurs le plus tôt possible dans le flot de conception en vue de réduire les coûts et les efforts liés à la vérification fonctionnelle. Dans le flot ESL, chaque niveau d'abstraction fournit un certain nombre de détails pour satisfaire un objectif de conception ou de vérification impliquant des nouveaux outils et langages. Arrivé au niveau RTL, le flot de conception ESL s'apparente au flot classique. Étant donné que le nouveau point d'entrée du flot de conception est un modèle SystemC et que le modèle de référence du sous flot de conception est au niveau transactionnel, il est crucial d'effectuer une vérification étendue des modèles SystemC-TLM avant de procéder aux raffinements.

1.2.4.1 Les techniques de vérification

Il existe plusieurs techniques de vérification. De façon générale, on peut identifier deux catégories : les méthodes de vérification dynamique (ou en simulation) et statique (ou formelle).

La vérification dynamique consiste à simuler le système en appliquant des stimuli au modèle du système appelé DUV (*Design Under Verification*). Le rôle du moteur de simulation est d'évaluer la réaction du DUV en réponse aux stimuli appliqués en entrée. L'environnement de simulation est souvent exécuté dans une machine hôte. Cette approche n'est pas exhaustive. Elle ne garantit pas l'exactitude du système mais elle permet de détecter efficacement les erreurs fonctionnelles. La rapidité, la facilité d'utilisation et l'efficacité de la simulation fait d'elle la technique de vérification prédominante en in-

dustrie. Mis-à-part la détection efficace des comportements incorrects, elle permet aussi l'identification des problèmes de performance, les incohérences ou les lacunes de la spécification elle-même. Cependant, la simulation ne garantit pas que le système se comporte correctement dans toutes les situations possibles. Ainsi, la qualité de la simulation dépend de la qualité des tests appliqués en entrée.

La vérification statique est une technique pouvant être utilisée en complément à la simulation. Au lieu de vérifier individuellement ou séquentiellement les états internes du système, cette technique souvent algorithmique, vise à prouver que le comportement désiré est respecté dans tous les états possibles du système. L'une des techniques formelles les plus répandues est le *Model Checking*. C'est une technique automatisée qui ayant le modèle d'états finis du système et une propriété logique, vérifie systématiquement si la propriété est respectée à chaque état du modèle[BZ08]. Cette technique offre ainsi une analyse exhaustive mais souffre du problème bien connu de l'explosion combinatoire des états[BZ08], et ne peut généralement être appliquée que sur des portions du système.

L'intersection de ces deux techniques est l'**ABV** (*Assertion Based Verification*)[MBP07] qui offre un bon compromis. Dans le contexte d'une simulation, l'ABV est considérée comme une technique semi-formelle car elle permet de spécifier formellement les propriétés qui seront vérifiées lors de la simulation. Le principe de cette technique consiste à faire appel à un langage formel logico-temporel tel que PSL¹³ [PSL05] ou SVA¹⁴[SV009] pour la spécification des assertions. Une assertion est une construction exécutable qui fournit à l'utilisateur un moyen de surveiller qu'une certaine propriété est respectée dans une implémentation et de reporter les erreurs dans le cas contraire. Cette propriété est une description formelle d'un comportement désiré ou interdit. Les assertions peuvent être intégrées dans le code de manière procédurale ou définies conjointement au code de manière déclarative. Dans les deux cas, elles permettent la détection des erreurs au plus proche de leur source et facilitent ainsi le processus de débogage. En effet, les assertions peuvent être transformées en "moniteurs" et intégrées dans le modèle à vérifier. Un "moniteur" ou "*checker*" est un composant passif chargé de surveiller le respect ou la violation de la propriété qu'il implémente ; il est destiné à être connecté ou intégré dans le DUV et donc décrit dans le même langage que celui-ci. Il faut noter que l'utilisation des assertions n'est pas restreinte à la vérification par simulation. Les moniteurs peuvent être synthétisés et utilisés lors de l'émulation sur FPGA¹⁵, une technique de vérification plus poussée, ou encore intégrés définitivement dans le circuit final pour la surveillance des fonctions hautement critiques.

En 1999, les constructeurs du processeur Cyrix M3, 3^{ième} génération ont démontré que 25% des erreurs ont été trouvées en ajoutant les assertions alors qu'ils estimaient que la

¹³Property Specification Language

¹⁴System Verilog Assertions

¹⁵Field Programmable Gate Array

simulation du modèle RTL était terminée[FKL04]. Grâce à l'ABV, le temps de débogage peut être réduit de 80% [Bai07]. De plus, les assertions sont très commodées, elles peuvent être incluses à tous les stades de la conception et sans contraintes de quantité. Toutes ces raisons expliquent l'intérêt croissant des industriels pour l'ABV. Historiquement, cette méthode a été largement appliquée aux modèles RTL. En 2001, Bob Bentley, le directeur du centre de vérification à Intel, a signalé que 400 erreurs du processeur Pentium4 ont été trouvées grâce aux assertions[FKL04].

À ce niveau, les outils de génération automatique de moniteurs à partir des assertions et les outils de conception et de synthèse offrent les supports nécessaires pour l'intégration et la synthèse des assertions conjointement au modèle. Parmi les principaux outils développés pour la génération automatique des moniteurs RTL, on cite : FoCs (*Formal Checkers*)[ABG⁺00] de IBM-Haifa¹⁶, MBAC[BZ05] et Horus[MAOB08]. Dans les deux premiers outils, les propriétés temporelles sont traduites en automates. Le dernier utilise une technique différente. À partir d'une spécification formelle des propriétés PSL, FoCs transforme chaque propriété en automate fini non déterministe. Afin d'être reconnus par les simulateurs, ces automates suivent un procédé de déterminisation. Le nombre d'états de l'automate déterministe peut alors être exponentiel en fonction du nombre de transitions non déterministes. Les moniteurs sont obtenus par la traduction des automates en code VHDL ou Verilog.

L'outil MBAC proposé par Boulé et Zilic est utilisé pour la génération de moniteurs RTL en Verilog à partir d'assertions PSL en passant par les machines à états. Cependant, une approche différente est appliquée permettant d'obtenir les automates optimisés et du code RTL synthétisable. Les moniteurs obtenus sont par conséquent plus petits et peuvent être utilisés en émulation. Les résultats de la comparaison de ces deux outils, rapportés dans [BZ05] montrent une nette amélioration de la fréquence de fonctionnement, du nombre de Flip-Flop et de LUTs (Look-Up-Table) par rapport aux moniteurs générés par FoCs.

De l'autre côté, Horus, l'outil développé par TIMA, utilise une approche innovante pour la génération de moniteurs VHDL à partir d'assertions décrites en PSL sans passer par des automates. L'outil s'appuie sur une bibliothèque de moniteurs élémentaires en VHDL dont chacun correspond à un opérateur PSL. Le parcours de l'arbre syntaxique de la propriété PSL permet de construire de manière linéaire le moniteur VHDL global en interconnectant les moniteurs élémentaires correspondant aux opérateurs de l'assertion. Ainsi, cette approche ne peut pas entraîner une explosion dans la taille du moniteur global. Le fonctionnement des moniteurs élémentaires ainsi que la méthode d'interconnexion ont été prouvés corrects par vérification formelle en utilisant l'assistant de preuve

¹⁶<http://www.research.ibm.com/haifa/projects/verification/focs/>

PVS¹⁷[MAB06]. De plus, les moniteurs VHDL obtenus sont directement synthétisables.

1.2.4.2 La vérification des modèles SystemC

Le déplacement de certains aspects de la vérification fonctionnelle vers les premières phases de conception a plusieurs avantages : à ce stade, les modèles sont plus faciles et rapides à simuler et les sorties sont plus faciles à interpréter. À un haut niveau d'abstraction, l'intérêt des assertions peut être double. En plus de leur rôle dans la vérification fonctionnelle, elles peuvent aussi être utilisées dès les premières phases de conception, pour la spécification initiale dans une démarche dite ABD (*Assertion Based Design*)[FKL04]. Cette nouvelle philosophie expose les propriétés en tête du processus de vérification et de conception. Selon cette logique, la spécification est représentée formellement par un ensemble de propriétés. La validation du modèle de référence doit être faite par rapport à cet ensemble de propriétés ce qui renforce la confiance en ce modèle. Pour assurer le respect de la spécification et la conservation des fonctionnalités demandées, les modèles raffinés doivent être validés en utilisant ce même ensemble de propriétés. Ainsi, de même que le modèle SystemC-TLM est considéré comme modèle de référence, les propriétés de haut niveau peuvent être considérées comme une référence et être raffinées concurremment avec le modèle. Afin de concrétiser cette démarche, il faut d'une part offrir un support pour l'ABV dès les hauts niveaux d'abstraction et d'autre part, inclure les assertions dans le processus de raffinement du modèle par l'intermédiaire d'une transformation automatisée des descriptions formelles.

1.3 Motivations et contributions

Malgré les progrès connus par les outils d'automatisation de la conception (EDA¹⁸), les outils de vérification n'ont pas connu la même progression. Gabriel Lezmi, vice président de Synopsys en Europe a écrit dans le magazine électronique "ElectronicsWeekly.com" que la vérification reste encore le plus grand goulot d'étranglement dans le flot de conception des SoCs actuels[Lez13]. Elle représente à elle seule plus que 50% de l'activité et engendre la limitation de la productivité et l'augmentation des coûts. Dans une interview accordée au magazine EE-Times en 2007, Laurent Ducousso¹⁹, a révélé que la part la plus importante de la vérification est liée au débogage. En effet, plus que 100.000 tests sont nécessaires pour la validation d'une IP[Duc07]. Cet avis est aussi appuyé par une étude faite en 2011 par MentorGraphics dont le résultat est reporté dans la Figure 1.5. Cette Figure montre que le débogage consomme en moyenne 32% de l'activité de vérification.

¹⁷Prototype Verification System, <http://pvs.csl.sri.com>.

¹⁸*Electronic Design Automation*

¹⁹responsable de la vérification à la division de STMicroelectronics à Grenoble

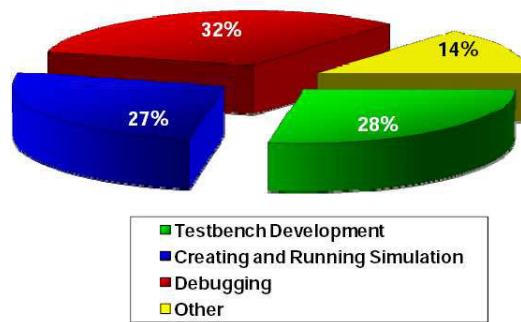


FIGURE 1.5 – Le temps moyen consommé par chaque tâche de vérification (source : [Fos11]).

Aujourd'hui les besoins des industriels s'orientent vers une méthodologie de vérification impliquant des outils et des méthodes permettant notamment l'automatisation du processus de débogage. L'utilisation des assertions est un excellent moyen pour réduire considérablement le temps de débogage. C'est une méthode qui a fait ses preuves auparavant au niveau RTL mais qui reste néanmoins sous exploitée aux niveaux d'abstraction supérieurs. L'avenir de la vérification et de la conception dans le flot ESL se dirige de plus en plus vers une démarche ABD. Actuellement, il y a des efforts qui poussent vers la standardisation des bibliothèques d'assertions au niveau transactionnel, au niveau RTL et même hybride mais il n'existe pas une solution qui permette le raffinement des assertions utilisées pour la validation du modèle de référence dans le modèle RTL et les modèles intermédiaires.

Les travaux menés dans le cadre de cette thèse se positionnent dans le cadre d'un flot de conception descendant basé sur SystemC-TLM et se focalisent sur l'ABV dynamique : l'objectif est d'établir un lien formel entre les assertions spécifiées au niveau transactionnel et celle du niveau RTL en procédant par raffinements semi-automatiques de façon à créer un flot de vérification conjoint au flot de conception du SoC. D'un point de vue pratique, ce flot pourra établir le lien entre la vérification au niveau SystemC-TLM telle qu'elle peut être réalisée avec l'outil ISIS[FP09][PF10] et la vérification au niveau RTL réalisable avec l'outil Horus. Le flot de vérification ciblé est représenté par la Figure 1.6.

1.3.1 Le raffinement des assertions TLM

Nous considérons le raffinement des assertions comme le processus de transformation d'une assertion décrite à un niveau d'abstraction vers un autre niveau d'abstraction tout en assurant la conservation de la propriété à surveiller. Cette transformation peut être réalisée en ajoutant les informations nécessaires relatives au nouveau niveau d'abstraction et en supprimant les informations devenues obsolètes de l'ancien niveau

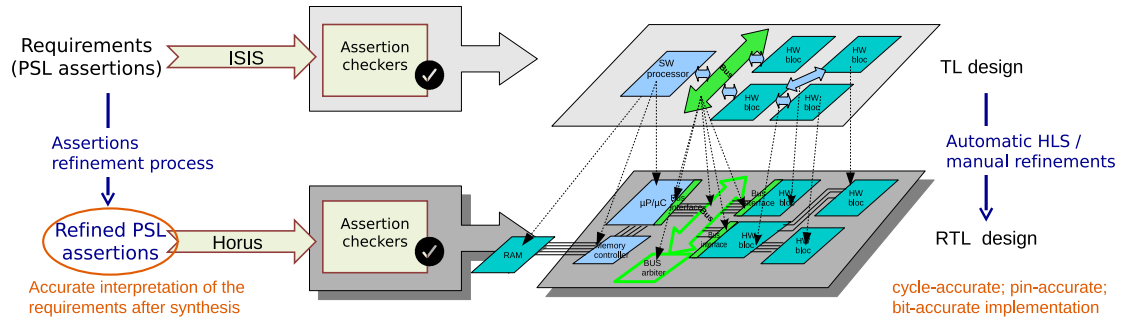


FIGURE 1.6 – Flot de vérification parallèle

d'abstraction[EESV07].

La principale difficulté du principe de réutilisation provient de la différence des méthodes de synchronisation et de communication aux niveaux TLM et RTL. Au niveau RTL, la synchronisation est basée sur des signaux d'horloge et de contrôle dédiés. Au niveau transactionnel, la synchronisation des échanges entre les composants est basée sur les événements. Elle est obtenue par des dépendances mutuelles entre les transactions et potentiellement par l'utilisation d'annotations temporelles ou de signaux asynchrones. Au niveau TLM, la communication est vue de façon atomique par des appels de fonctions atomiques qui permettent l'échange des transactions. Au niveau RTL, le niveau de granularité est plus élevé et les appels de fonctions de communication sont traduits en un ensemble de signaux échangés selon un protocole de communication particulier. Ainsi, plusieurs détails temporels de synchronisation doivent être pris en considération. Par conséquent, une utilisation directe des propriétés TLM n'est pas possible pour les niveaux moins abstraits. À titre d'exemple, la Figure 1.7, décrit une opération d'écriture simple

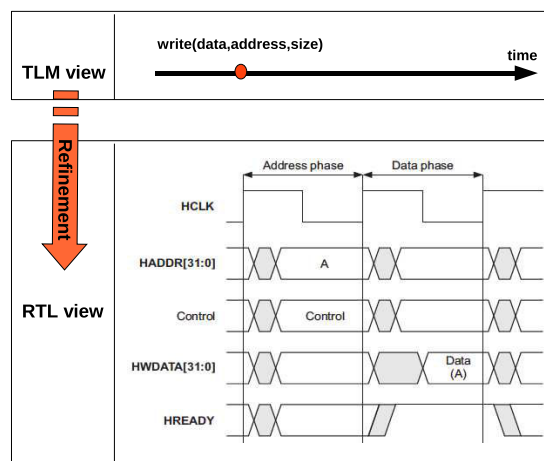


FIGURE 1.7 – La disparité des domaines temporels entre TLM et RTL

d'un bus AHB²⁰ représentée au niveau TLM et au niveau RTL. Au niveau transactionnel,

²⁰Advanced High-performance Bus (https://web.eecs.umich.edu/~prabal/teaching/eecs373-f11/readings/ARM_IHI0033A_AMBA_AHB-Lite_SPEC.pdf)

cette opération se traduit par un appel à la fonction “**write**” ayant comme paramètres : les données à écrire (**data**), l’adresse de l’écriture (**address**) et la taille des données(**size**). Selon le protocole de communication implémenté au niveau RTL, cette même opération est décrite par un ensemble de signaux de contrôle et de données synchronisés par une horloge (**HCLK**). Ainsi, l’appel de fonction atomique est traduit par une séquence de signaux étalés sur le temps. Ici par exemple, deux cycles d’horloges sont nécessaires pour accomplir une opération d’écriture simple : sur le premier cycle, les signaux de contrôle sont positionnés et l’adresse est transmise, et sur le second cycle, la donnée est envoyée. Le protocole peut être bien plus complexe dans d’autres types de transferts que dans cet exemple simple. L’enjeu est alors de fournir une méthode de raffinement qui respecte les particularités de chaque niveau tout en conservant la propriété surveillée, et de placer cette méthode dans un processus de transformation automatisé permettant la génération d’assertions hybrides et au niveau RTL à partir des assertions TLM utilisées pour la validation du modèle de référence.

1.4 Organisation du document

Ce manuscrit est structuré en sept chapitres incluant la présente introduction et une conclusion générale. Le deuxième chapitre est une introduction du contexte général de notre étude. Il se base sur la vision idéalisée du flot de conception pour mettre l’accent sur les différences entre les niveaux d’abstraction RTL et transactionnels. Cela nous permettra de présenter les méthodologies, outils et langages associés à chaque niveau et d’introduire la problématique de raffinement.

Le troisième chapitre présente, dans un premier temps, une introduction du langage de spécification d’assertions PSL. Dans un second temps, un état de l’art des stratégies et des méthodes de vérification à base d’ABV est présenté pour les descriptions aux niveaux RTL et TLM. L’accent sera mis sur les solutions de vérification utilisées par la suite en justifiant les choix portés.

Les chapitres suivants se focalisent sur l’explication de notre solution de raffinement des assertions qui vise à résoudre les limitations des travaux proposés dans ce sujet.

Dans le quatrième chapitre, premièrement, nous nous focalisons sur l’explication des différentes problématiques liées au raffinement des assertions afin de souligner la nécessité d’apporter une solution automatisable. Ensuite, une énumération des principaux travaux effectués dans ce contexte sera faite pour mettre le point sur les apports de la méthode proposée. Enfin, nous concluons le chapitre par une introduction des règles formelles de raffinement à la base de notre solution.

Le cinquième chapitre détaille la mise en œuvre de tous les concepts liés au raffinement des assertions. Il commence par une présentation de l’outil de raffinement permettant

l'automatisation de la génération des assertions raffinées et d'achève sur une explication des différentes extensions faites sur l'outil ISIS en relation avec la problématique de raffinement.

Afin d'illustrer l'intérêt de la solution proposée, des d'expérimentations sur plusieurs assertions feront l'objet du sixième chapitre. Ce chapitre se termine par une analyse, à la lumière des résultats obtenus, des apports et des limitations de l'approche développée et une discussion sur les améliorations qui peuvent être apportées.

Enfin, la conclusion résumera les travaux exposés dans ce mémoire et présentera les principales perspectives envisagées.

Langages de description et niveaux d'abstraction

Sommaire

Introduction	52
3.1 La spécification à base d'assertions :PSL	52
3.1.1 Les opérateurs FL	54
3.1.1.1 Le sous ensemble simple PSL_{ss} (<i>Simple Subset</i>) . . .	56
3.1.1.2 Les opérateurs faibles et forts	57
3.1.1.3 Les niveaux de satisfaction des FL	57
3.1.2 La sémantique de PSL	58
3.1.2.1 La sémantique des opérateurs temporels FL	59
3.1.2.2 La sémantique des SERES	62
3.2 La vérification dans les flots de conception	64
3.2.1 La vérification au niveau RTL : Horus	65
3.2.2 La vérification au niveau TLM	67
3.2.2.1 Les approches au niveau <i>Cycle Accurate</i>	67
3.2.2.2 Les approches au niveau transactionnel	69
3.2.2.3 L'outil ISIS	75
3.2.2.4 L'utilisation des SERES et de l'opérateur next	81
3.3 Bilan	83

“What you do up front determines how difficult and costly your development flow will be”.

Gabe Moretti in “Increasing the level of Abstraction of IC design” on EDACafe.

Introduction

Dans ce chapitre, nous allons suivre l’histoire de l’évolution des flots de conception des SoCs dans le but de mieux caractériser les différents niveaux d’abstraction. On commence alors par la description des niveaux de modélisation utilisés dans le flot de conception classique. En deuxième lieu, on se positionne dans le cadre d’un flot de conception ESL descendant basé sur SystemC-TLM tel que celui présenté dans le premier chapitre (*c.f.* section 1.2.2) pour identifier les principaux niveaux d’abstraction supportés. Nous rappelons que l’avantage de cette approche est que les concepteurs peuvent prendre les décisions et faire les choix déterminants à un niveau où les détails inutiles sont négligés. Ceci leur permet de manipuler une version simplifiée du système complexe, d’introduire de manière incrémentale les détails relatifs à une vision du modèle afin de servir à chaque fois un objectif de conception ou de vérification distinct. Nous proposons de suivre l’évolution des modèles selon ce processus afin d’expliquer les concepts liés à chaque étape. Nous commencerons alors par une étude des langages et des standards de modélisation utilisés au niveau RTL selon le flot de conception classique afin de remonter ensuite dans les niveaux d’abstraction jusqu’à atteindre les niveaux système du flot ESL. Un flot ESL détaillé sera donné en conclusion de cette analyse incluant les aspects raffinés et les détails introduits pour passer d’un niveau d’abstraction à un autre. Cette approche vise à mieux comprendre le passage vers le flot de conception basé sur ESL, expliquer les caractéristiques et l’utilité de chaque niveau et identifier les outils et les langages utilisés à chaque étape.

2.1 Les niveaux d’abstraction du flot RTL

La description *Register Transfer Level* (RTL) se situe entre une description purement comportementale du circuit désiré et une description purement structurelle. C’est un niveau d’abstraction supérieur au niveau portes logiques qui permet de présenter les composants du système sous la forme de registres (*flip-flops*, *latches*) et les communications entre ces derniers en termes d’opérations logiques entre des signaux d’interconnexion d’où la dénomination “RTL”. C’est la vue abstraite la plus détaillée du système complet et la qualité du circuit final issu des étapes suivantes dépend de la qualité de ce modèle. La qualité d’un tel modèle est fonction de plusieurs critères : sa correction, le degré de réutilisation, sa compatibilité aux outils de synthèse de bas niveau et sa correspondance aux spécifications fonctionnelles et aux exigences.

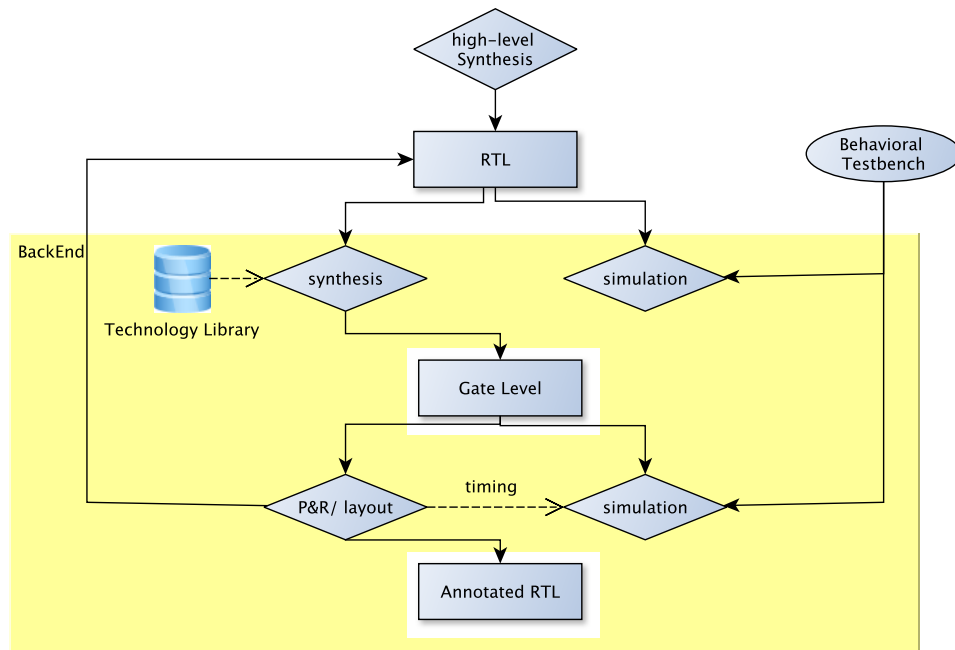


FIGURE 2.1 – Le flot de conception partant du modèle RTL

Les langages HDL¹ tels que VHDL et Verilog sont utilisés pour décrire les circuits à ce niveau. L’objectif de cette modélisation est l’automatisation de la synthèse des modèles et l’optimisation des circuits en termes de temps, de surface et de consommation d’énergie. La Figure 2.1 montre le flot de conception commençant par le modèle RTL comportemental. Selon cette figure, en premier temps, le modèle doit être simulé afin de s’assurer de sa correction et de sa conformité par rapport à la spécification. Cette étape est dite “validation”.

Dans un deuxième temps, le code validé est synthétisé en utilisant les bibliothèques de composants primitifs d’une certaine technologie afin de générer la “netlist” et le code de type “portes logiques”. Ensuite, vient l’étape du placement/routage du circuit (ou du *layout* s’il s’agit d’un ASIC²). Une fois le circuit routé, les temps de propagation sont extraits et le système est resimulé pour vérifier que le placement/routage est suffisamment optimisé pour respecter les contraintes de temps. Si ces contraintes ne sont pas respectées, le processus est relancé pour tenter une optimisation. Ce processus itératif est renouvelé un nombre de fois suffisant pour satisfaire les contraintes. Dans le cas où elles ne le sont pas, l’outil de synthèse déclare alors qu’il ne peut router le circuit de façon à tenir les contraintes, et donne les chemins critiques.

Nous utilisons l'exemple d'un bus wishbone[ope10] codé en VHDL proposé dans [Sch13]

¹Hardware Description Language

²Application-Specific Integrated Circuit, pour circuit intégré propre à une application. Les ASICs peuvent aussi être utilisés comme des IPs

pour illustrer les caractéristiques générales du niveau RTL.

Le bus wishbone peut être défini pour plusieurs topologies. Celle implémentée ici est dite “crossbar”. Les interfaces du bus ainsi que la signification des signaux sont disponibles dans [ope10] pour les composants *master* et *slave*. Comme l’illustre la Figure 2.2, le bus se comporte comme un switch organisant la communication entre plusieurs *masters* et *slaves*. Les traits discontinus de la figure dénotent des liaisons possibles entre les composants. Chaque liaison met en relation un *master* et un *slave*.

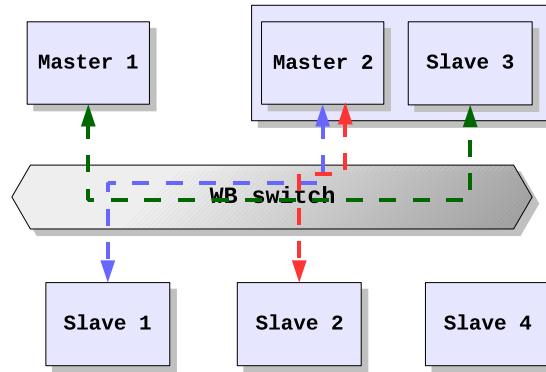


FIGURE 2.2 – La topologie crossbar des bus de communication (source :[ope10])

L’architecture du bus est illustrée par la Figure 2.3. Le bus se compose d’un multiplexeur pour les masters géré par un arbitre et d’un dé-multiplexeur pour les *slaves* géré par un décodeur d’adresses.

Le rôle de l’arbitre est d’implémenter un algorithme d’arbitrage permettant d’organiser l’accès des différents masters au bus.

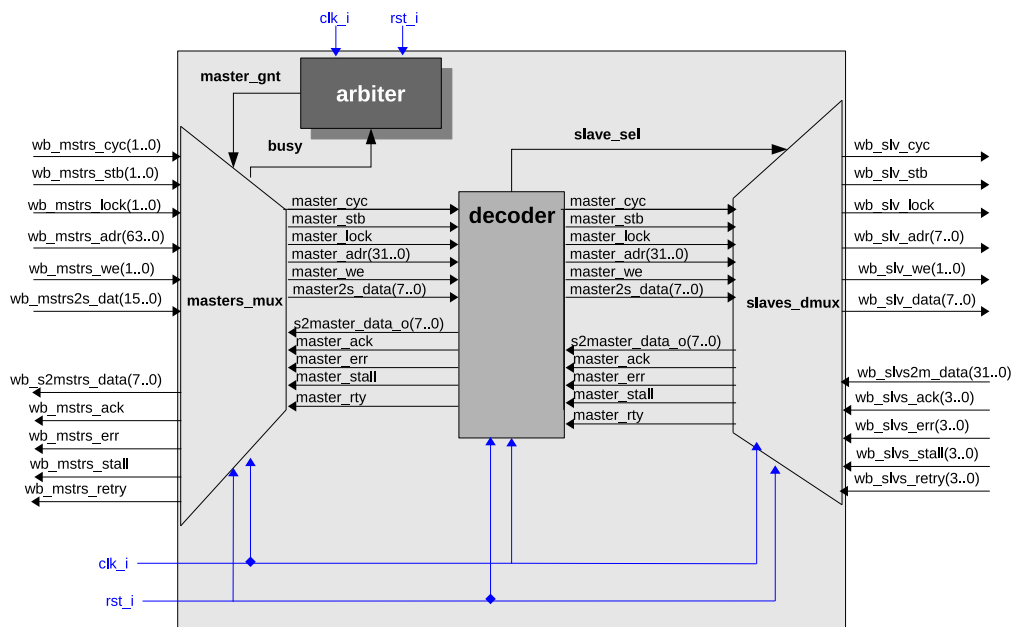


FIGURE 2.3 – La représentation de l’architecture du bus wishbone

La Figure 2.3 décrit les connexions pour un bus pouvant relier au plus deux *masters* et quatre *slaves*. La figure illustre qu'à ce niveau de description, toutes les communications sont faites par des signaux. Il faut noter aussi que les interfaces des composants sont identiques aux interfaces décrites dans la spécification du bus wishbone.

L'extrait de code donné dans le programme 2.1 contient la déclaration de l'entité et de l'architecture de l'arbitre. L'entité donne une vue externe du composant décrivant les ports d'entrée et de sortie exposés à son environnement. La description RTL est dite *pin-accurate* car la description des interfaces des modules codés au niveau RTL reflète l'interface des composants matériels de l'implémentation.

Le composant décrit possède trois ports d'entrée `clk_i` (l'horloge), `rst_i` (*reset*), `busy_i` et un port de sortie `gnt_o`. Les types des ports indiquent le type des signaux susceptibles de se connecter à chaque port. Nous soulignons qu'au niveau RTL, les ports et les signaux sont tous au niveau bit ou vecteur de bits d'où l'appellation *bit-accurate*. Le port de sortie, par exemple, est défini comme un vecteur de bits de taille paramétrable.

```
-- wb_arbiter entity
entity wb_arbiter is
generic (
    mstr_bits    : natural := 1
);
port (
    -- interface declaration
    clk_i        : in  std_logic;
    rst_i        : in  std_logic;
    busy_i       : in  std_logic;
    gnt_o        : out std_logic_vector((mstr_bits - 1) downto 0)
);
end wb_arbiter;

-- wb_arbiter architecture
architecture Behavioral of wb_arbiter is
    signal cnt : unsigned((mstr_bits - 1) downto 0) := (others => '0');
begin
    process -- explicit declaration of the VHDL process
    begin
        -- process synchronisation on clock posedge events
        wait until rising_edge(clk_i);
        if (rst_i = '1') then
            cnt <= (others => '0');
        elsif (busy_i = '0') then
            cnt <= cnt + 1;
        end if;
    end process;
end architecture;
```



```

--implicit process
gnt_o    <= std_logic_vector(cnt);
end Behavioral;

```

PROGRAMME 2.1 – Déclaration de l'entité *arbiter*

L'architecture est une vue interne du composant. Pour les composants hiérarchiques³ tels que le bus wishbone, une description structurelle de l'architecture sous la forme d'un chemin de données (*datapath*⁴) est préférable car elle donne une vue modulaire plus globale sur le système et favorise la réutilisation des composants.

Dans le cas de l'arbitre, la description de l'architecture est comportementale : la fonction d'arbitrage (ici un tourniquet) est implémentée par un processus synchronisé sur les fronts montants de l'horloge, comme l'exprime l'instruction “**wait until** rising_edge(clk_i)”. Ce comportement dit *cycle-accurate* est caractéristique d'une description au niveau RTL. Nous notons aussi l'existence d'un mécanisme d'initialisation synchrone du composant assuré par le signal booléen de reset **rst_i** actif à l'état haut.

En conclusion, RTL est le niveau de modélisation le plus proche de l'implémentation matérielle. En raison de son niveau de précision élevé (*cycle-accurate*, *bit-accurate*, *pin-accurate*), la vitesse de simulation d'un modèle RTL est réduite, de l'ordre de [1-100kIPs], et limite considérablement la capacité de conception et de vérification des SoCs complexes.

2.2 Les niveaux d'abstraction dans le flot ESL

L'implémentation de la méthodologie de conception ESL nécessite de nouveaux outils et langages qui supportent la modélisation au niveau système. Le logiciel et le modèle fonctionnel sont souvent décrits en C/C++. La description des modèles matériels au niveau système nécessite en revanche de faire appel à des langages de description du matériel de haut niveau appelés SLDL⁵ tels que SystemC et ses caractéristiques TLM.

2.2.1 Les langages de modélisation de haut niveau

2.2.1.1 La bibliothèque SystemC

SystemC est un langage⁶ de conception du matériel né du besoin des concepteurs de modéliser les SoCs au niveau système[GLMS02]. Il est néanmoins important de préciser qu'il n'est pas lié à un processus de conception particulier (ascendant ou descendant). En effet, il est bâti sur le principe que tous les flots de conception sont itératifs et que de ce fait il est fréquent que les différents composants d'un même système soient à des niveaux

³composé d'autres sous composants

⁴instances de composants reliées par des signaux

⁵System Level Design Language

⁶même si SystemC est une bibliothèque C++, il est souvent considéré comme un langage

d'abstraction différents. Ceci se traduit au niveau du langage par deux concepts clés : la programmation modulaire et la séparation des concepts. La programmation modulaire est une caractéristique générale des langages orientés objet. SystemC est fondé sur le langage orienté objet C++ et l'hérite ainsi de ce dernier. Le principe de séparation des aspects est un concept qui puise ses origines dans la programmation fonctionnelle des logiciels. Il a été introduit par souci de simplification des programmes complexes et vise à identifier les parties fonctionnellement indépendantes du programme afin de les traiter séparément. C'est pour cette même raison que les langages de conception du matériel complexe font appel à ce principe. La combinaison de ces deux concepts dans un seul langage favorise la conception de systèmes flexibles et facilement réutilisables. Ces deux concepts se reflètent dans la structure même de cette bibliothèque. En effet, SystemC est construit en plusieurs couches dont la base est un moteur de simulation (*kernel*) événementiel performant. Ce moteur a une vision abstraite des événements et des processus qu'il orchestre indépendamment de leurs contenus. Le cœur de SystemC est composé de ce moteur de simulation et d'un ensemble de classes et de macros pour la description structurelle et fonctionnelle du matériel. On y trouve les modules, les processus, les ports, les interfaces, les canaux de communication et les événements.

Les modules

Un module est l'unité de construction de base d'un modèle SystemC. C'est l'équivalent d'un composant dans l'implémentation. Concrètement, un module est une classe C++. Il est typiquement composé de ports le reliant à d'autres modules de son environnement, de processus concurrents qui exécutent ses fonctionnalités et éventuellement d'autres sous modules. Un module peut être instancié une ou plusieurs fois pour représenter des composants aux comportements identiques au sein de la même architecture. Dans le cas d'une structuration hiérarchique, les échanges entre les sous modules ne sont pas exposés à l'environnement extérieur du module englobant comme le montre la Figure 2.4. Ce mode de construction modulaire permet une grande flexibilité architecturale.

Dans un modèle RTL, la communication entre les composants se fait exclusivement à travers des signaux. SystemC offre cette possibilité afin de permettre la modélisation de systèmes au niveau signal mais y ajoute d'autres structures d'interconnexion plus abstraites pour la modélisation au niveau système telles que les canaux, les interfaces, les ports.

Les canaux

Les canaux représentent le support de transmission des données entre les différents modules. La communication est une opération qui met en jeu deux modules dont l'un est l'émetteur des données appelé aussi "*initiator*" ou "*master*" et l'autre est le récepteur appelé "*target*" ou "*slave*". Toutefois, un canal peut assurer la communication entre

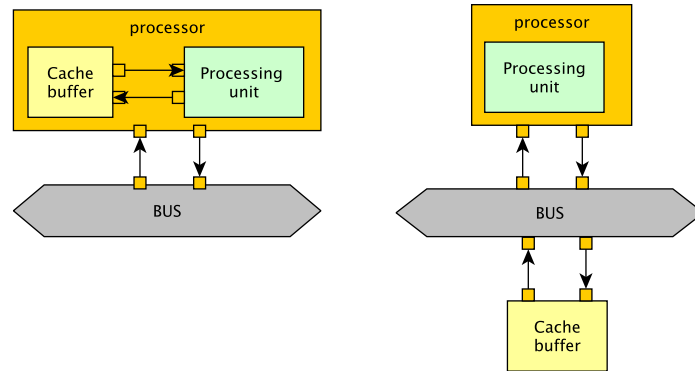


FIGURE 2.4 – Différence entre un module hiérarchique et un module simple

plusieurs modules et poser des restrictions d'accès. Si on considère que la communication est faite par un signal, le canal peut exiger qu'un seul module à la fois y accède en écriture tandis que plusieurs peuvent y accéder en lecture. Le rôle du canal est alors de fournir une implémentation qui décrit le déroulement des opérations de transmission. Gajski et al. [CG03] étaient parmi les premiers à définir le concept de canaux de communication pour permettre la séparation des aspects dans les langages de conception du matériel de haut niveau (SLDL).

SystemC définit deux classes de canaux : primitifs et hiérarchiques. Comme leur nom l'indique, les canaux primitifs sont atomiques. Contrairement aux canaux hiérarchiques, ils ne contiennent pas d'autres structures SystemC et offrent ainsi une abstraction du mécanisme de communication. Le signal `sc_signal<T>` ("T" le type du signal) est un canal primitif. Les canaux hiérarchiques en revanche, sont composites. Ils peuvent contenir des modules et des processus pour modéliser des structures de communication plus développées comme un bus avec un arbitre et une unité de décodage d'adresses. Ainsi, un canal hiérarchique peut être vu comme une vue plus détaillée d'un canal complexe.

Les interfaces

L'interface spécifie les méthodes que le canal doit implémenter. Ceci est concrétisé par une classe spéciale qui expose uniquement la signature de ses méthodes. Cette interface peut être implémentée par plusieurs canaux offrant chacun une définition différente de ces

méthodes. La conception basée sur les interfaces (*interface-based design*) [RSV97] illustrée par la Figure 2.5 systématise le principe de séparation des aspects. Dans SystemC l'interface est l'intermédiaire donnant l'accès au canal de communication. Par exemple, les interfaces correspondant à un signal sont “`sc_signal_in_if<T>`” et “`sc_signal_inout_if<T>`” avec “`T`” le type du signal. La première interface est relative aux signaux entrants et offre uniquement la fonction abstraite “`read()`” permettant d'accéder au contenu du signal. La deuxième interface hérite de la première et offre en plus la fonction “`write()`” permettant l'accès en écriture.

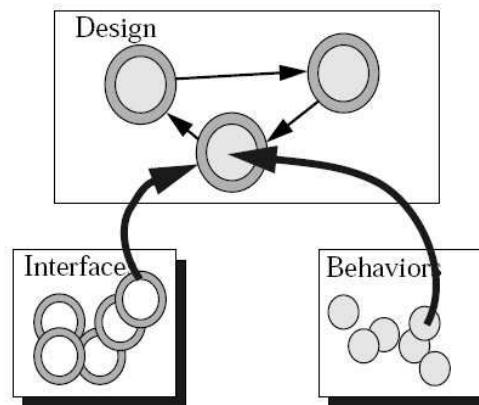


FIGURE 2.5 – conception basée sur les interfaces (source : [RSV97])

Les ports

Un port est un objet lié au module. Il est chargé des interactions du module avec les canaux. Ainsi, le port est directement connecté à l'interface du canal. Cette connexion lui permet d'accéder aux méthodes du canal exposées par l'interface. SystemC fournit la classe “`sc_port<interface,N>`” pour la création des ports avec en premier paramètre l'interface à laquelle correspond ce port et en deuxième paramètre le nombre maximal d'interfaces attachées au port (par défaut $N = 1$). Dans le cas où $N > 1$ on parle d'un multi-port. Par exemple, dans le cas de l'interface “`sc_signal_inout_if<T>`” qui expose une méthode de lecture et une méthode d'écriture, le port p associé est déclaré comme suit : `sc_port<sc_signal_inout_if<T> > p`. L'accès aux méthodes du canal est fait par une surcharge de l'opérateur “`→`” qui retourne au port un pointeur vers l'interface. Ainsi, si on reprend notre exemple, l'expression “`p→read()`” permettra de lire la valeur du signal sur le port p .

La combinaison port-interface-canal correspond à ce qu'on appelle *Interface Method Call* (IMC) illustrée par la Figure 2.6. La figure décrit un module *initiator* demandant une lecture dans un *target* via son port p . Le *target* est un canal qui expose à son interface de transport la méthode `read()`. L'*initiator* n'a accès qu'à l'interface. Il est alors possible de changer l'implémentation de la méthode tout en conservant la même interface.

C'est un moyen efficace notamment pour le raffinement des supports de communication. Tandis que l'interface reste invariante, son implémentation peut être incrémentalement raffinée indépendamment de son environnement. De plus, l'*Interface Method Call* permet une meilleure exploration des architectures. En effet, il est possible, par exemple, d'interchanger plusieurs supports de communication sans avoir besoin de faire des modifications majeures dans le modèle.

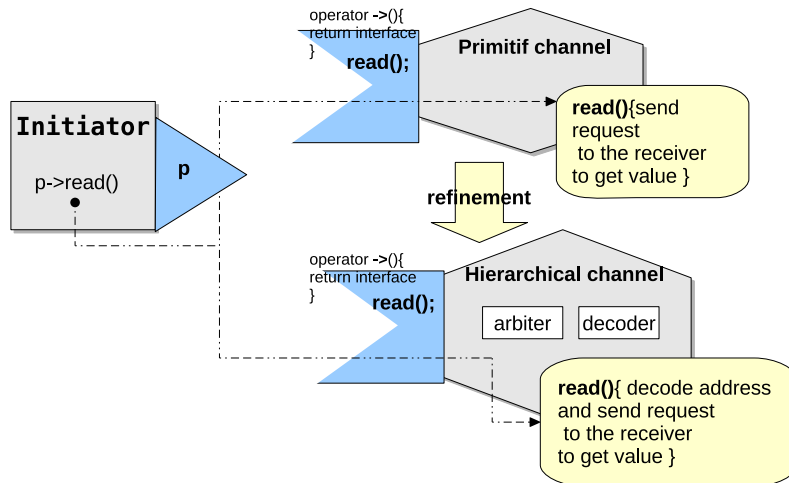


FIGURE 2.6 – *Interface Method Call* et raffinement du support de communication

Selon le rôle du module dans la communication (*initiator* ou *target*), le port peut être un “`sc_port<interface>`” ou un “`sc_export<interface>`”. Le premier est pour initier des transactions, le second est pour en recevoir.

Les processus

Un processus est l'unité fonctionnelle de base des modules SystemC. Contrairement aux fonctions habituelles utilisées pour modéliser des comportements séquentiels, les processus fournissent un moyen pour simuler les comportements concurrents dans les architectures matérielles. Analogiquement aux processus VHDL, les processus SystemC peuvent avoir une liste de sensibilité décrivant les événements auxquels ils doivent réagir. De plus, certains types de processus (les *threads*) peuvent suspendre leur exécution en attente d'un ou d'un ensemble d'événements au moyen de la fonction `wait()`. Ainsi, un processus demeure actif jusqu'à ce qu'il rencontre un “`wait()`” ou un “`return()`”.

Les événements et la synchronisation

Jusque là, nous nous sommes implicitement appuyés sur une définition intuitive du terme “événement”. SystemC donne une définition précise de ce terme : c'est un objet

de type “**sc_event**” utilisé comme un moyen de synchronisation des processus. Une instance d'un processus peut démarrer ou reprendre son exécution suite à l'occurrence d'un événement auquel il est sensible[SC005]. Un événement est ainsi porteur d'information. Cependant il n'a ni durée, ni valeur. Dans la plupart des cas, le processus source notifie⁷ l'événement d'un changement de son état pour permettre au processus destination sensible à cet événement de réagir en réponse à ce changement. Cette notification peut avoir plusieurs occurrences discrètes au cours de la simulation. Le moteur de simulation de SystemC (*kernel*) organise l'exécution des processus en fonction de la notification des événements. La plupart des langages de modélisation du matériel (SystemC, VHDL, Verilog) utilisent un moteur de simulation (*kernel*) basé sur les événements. Contrairement à une simulation continue, où le temps est découpé en tranches égales et le temps avance systématiquement à chaque tranche, dans la simulation basée sur les événements discrets, le temps avance lorsque l'état du système change suite à la notification d'un événement. C'est pour cette raison qu'un moteur de simulation événementiel est souvent plus rapide qu'un moteur de simulation continue.

La synchronisation par une horloge : L'horloge sert à synchroniser les processus sur des événements périodiques, le modèle résultant fournit une précision temporelle au niveau cycle (*cycle-accurate*). En SystemC, les horloges sont considérées comme des canaux hiérarchiques. La déclaration de l'horloge se fait par la création d'un objet du type “**sc_clock**” qui peut être configuré via des paramètres indiquant entre autres la période de l'horloge, le temps passé à l'état haut et à l'état bas et la nature du premier front. La classe fournit aussi un ensemble de services permettant par exemple d'accéder à la valeur de l'horloge (**read()**), de détecter la notification d'un front (**posedge_event()**, **negedge_event()**) ou d'un changement de valeur (**event()**). La synchronisation est assurée par ces événements.

La Figure 2.7 donne un aperçu général des interactions entre les modules de l'utilisateur et le moteur de simulation SystemC permettant de créer une spécification exécutable. L'exécution d'une description SystemC passe par deux phases : la phase d'**élaboration** pendant laquelle les parties et la hiérarchie du modèle sont créés, les ports et les exports sont reliés aux canaux et les listes de sensibilités sont établies et la phase de **simulation** pendant laquelle l'ordonnanceur du moteur de simulation de SystemC organise l'exécution des processus en fonction de la notification des événements.

Le modèle de temps

⁷L'acte d'indiquer une modification à un événement

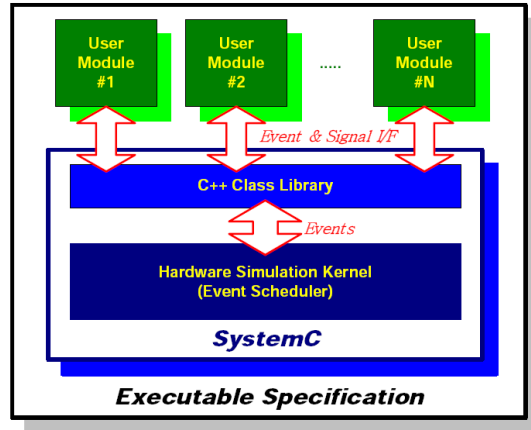


FIGURE 2.7 – Le mécanisme de synchronisation dans SystemC

SystemC utilise un modèle de temps absolu, discret : le temps est représenté par un couple “(entier,unité)”. La première composante est un nombre entier non signé sur 64 bits. SystemC définit un type énuméré des unités de temps, “`sc_time_unit`” allant de la femtoseconde (SC_FS) à la seconde (SC_SEC). Par défaut, la résolution du temps est 1 picoseconde (10^{-12} s). La déclaration d’une variable de temps est faite par l’instanciation de la classe “`sc_time`”. SystemC fournit aussi un moyen de connaître l’instant de déroulement d’une action ou d’un traitement durant la simulation via la fonction “`sc_time_stamp()`”.

2.2.1.2 La modélisation avec la bibliothèque SystemC-TLM

Dans cette partie, on commencera par une description des principales composantes de la bibliothèque SystemC-TLM. Un exemple de plateforme sera donné par la suite pour illustrer les notions présentées.

La bibliothèque TLM est un ensemble de classes C++ construites en amont de SystemC et offrant les mécanismes nécessaires pour implémenter les concepts de la méthodologie TLM. Ces mécanismes permettent la construction des schémas de communication les plus usuels, notamment le bus *memory-mapped*. TLM s’appuie sur la description architecturale de SystemC. Les structures supplémentaires proposées ne touchent qu’aux aspects de communication.

La bibliothèque TLM est le résultat du travail du TLM Working Group⁸. La première version de cette bibliothèque [OSC05] a été créée en 2005 après la publication de la version 2.0 de SystemC (2001). Elle contenait un ensemble de classes pour la standardisation des interfaces de communication bloquantes et non bloquantes. Cependant, le principal inconvénient de cette version est l’absence d’une structure de donnée standardisée de l’objet décrivant la transaction. Chaque concepteur devait fournir sa propre structure, ce

⁸Groupe de travail créé par OSCI et qui regroupe des représentants de plusieurs industriels tels que : STMicroelectronics, MentorGraphics, Cadence, ARM.

qui, par conséquent, limite considérablement l'interopérabilité des modèles implémentés. Le nombre réduit d'interfaces et de mécanismes de communication proposés est une autre limitation de cette première version.

L'objectif de la deuxième version de cette bibliothèque sortie en 2008 est de fournir une architecture de transport complète et standardisée pour une meilleure interopérabilité et une optimisation de la vitesse de la simulation. Selon le manuel de référence de la TLM-2[OSC09], la bibliothèque est construite en plusieurs couches et assure la compatibilité ascendante avec la TLM-1. Toutefois, il est fortement recommandé d'utiliser les éléments de la couche interopérabilité de la nouvelle version. La Figure 2.8 montre que cette couche est composée d'un ensemble d'interfaces de communication (ou de transport) qui forment le cœur de la TLM-2 et d'un ensemble de structures de communication telles que : les sockets, le generic payload, le base protocol et le global quantum plus spécifiques pour la modélisation des bus *memory-mapped*.

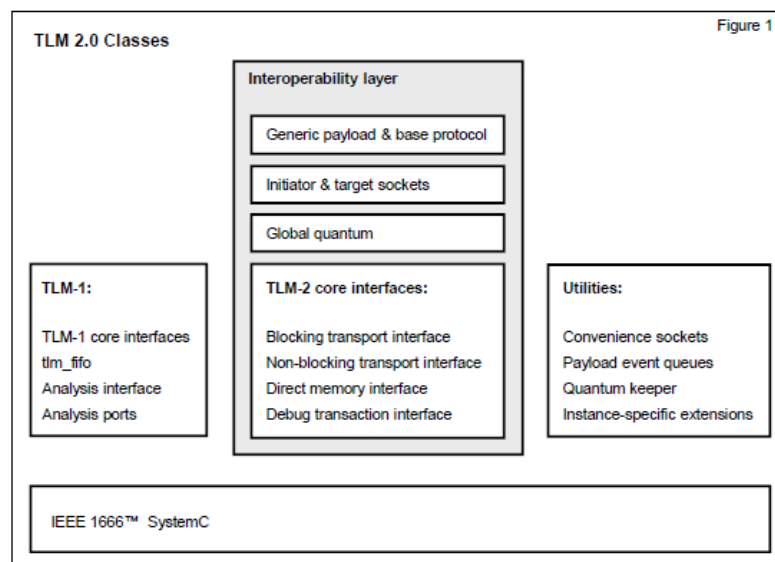


FIGURE 2.8 – La structure de bibliothèque TLM-2(source :[OSC09])

Selon cette bibliothèque, la transaction est faite entre un module *initiator* qui encapsule ses données dans un objet générique nommé *payload* et l'envoie en argument de la fonction de transport à un autre module *target* qui le reçoit, le decode, l'exécute et renvoie vers l'*initiator* éventuellement une réponse. Si on se réfère à une modélisation SystemC, on dira que chaque module *initiator* (et inversement pour le *target*) doit avoir par conséquent un objet *sc_port* pour l'envoi de la transaction et un objet *sc_export* pour la réception. TLM-2 encapsule ces deux structures dans un connecteur particulier dit *socket*. Les sockets de la TLM-2 regroupent toutes les interfaces de transport de la couche interopérabilité. Grâce à cette structure, le modèle de communication devient flexible permettant, à moindre coût et en toute transparence, le passage d'une interface de transport

à une autre. Du côté de l'*initiator*, on parle d'un `tlm_initiator_socket` et symétriquement, du côté du *target*, d'un `tlm_target_socket`. Dans son chemin de l'*initiator* vers le *target* une transaction peut passer par un ou plusieurs modules qui représentent dans la plupart des cas des canaux hiérarchiques. Ces composants qui ne sont pas la destination finale de la transaction peuvent affecter le contenu de la transaction (décodage d'adresse, arbitrage, etc.) lors de son acheminement et représentent des composants d'interconnexion.

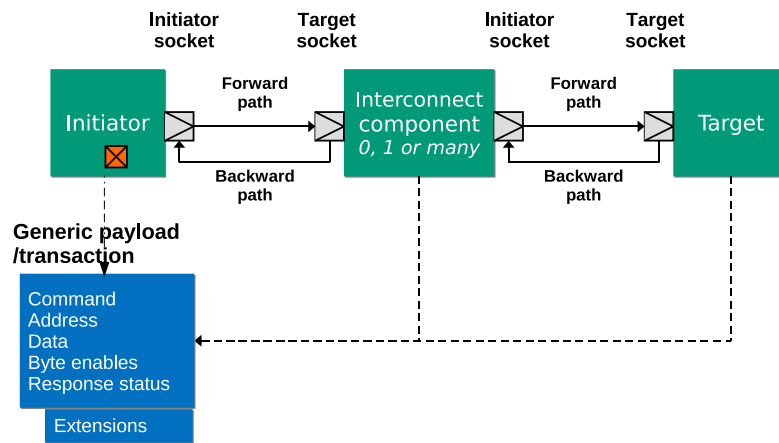


FIGURE 2.9 – Le schéma d'interconnexion dans TLM-2 (source :[Ayn09])

Selon ce schéma, illustré par la Figure 2.9, le chemin d'envoi est composé de tous les appels de fonctions de transport permettant d'acheminer la transaction de l'*initiator* vers le *target* final. À l'intérieur, un module d'interconnexion joue le rôle d'un *target* vis à vis du module précédent et d'*initiator* par rapport au module suivant et doit de ce fait avoir à la fois un *target socket* et un *initiator socket*. Comme le montre la Figure 2.9, l'objet transaction du type `tlm_generic_payload` est une structure englobant les données passées dans une transaction contenant le type de l'opération (*read/write*), l'adresse, les données et d'éventuelles informations supplémentaires (masquage, état de la réponse). Son utilisation n'est pas obligatoire mais elle peut s'avérer particulièrement utile dans la modélisation de certaines interfaces des bus et de nombreux protocoles de communication. Cette classe peut être étendue par héritage si nécessaire. De plus, TLM-2 a prévu un mécanisme d'extension permettant d'inclure plus facilement des données supplémentaires dans cette structure.

Le modèle de communication de TLM : Conformément au principe de l'*Interface Method Call* évoqué précédemment et illustré dans le cas de SystemC par la Figure 2.6, dans la bibliothèque TLM-2, en raison des améliorations introduites dans les structures

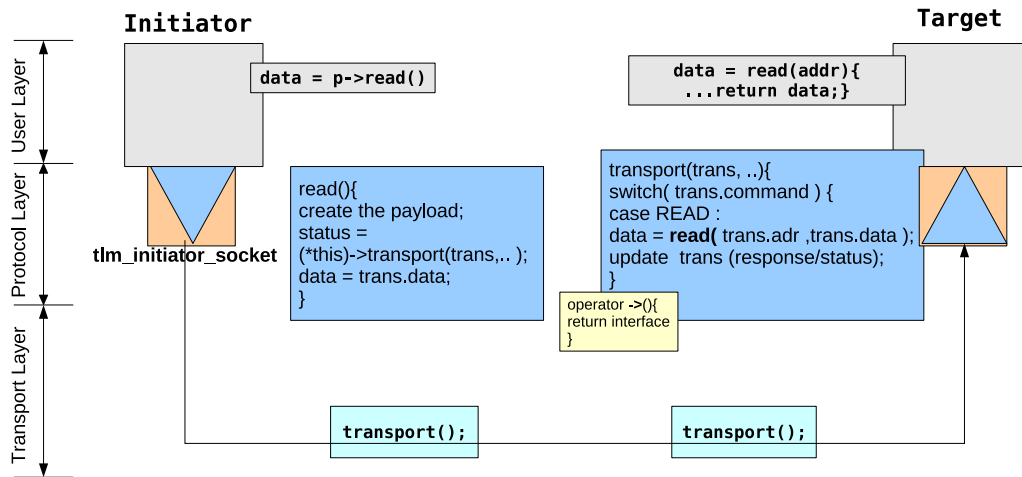


FIGURE 2.10 – le déroulement de l'Interface Method Call dans TLM-2

de communication, le déroulement d'une opération de communication se traduit selon un modèle de trois couches[RSPF05]. Toutefois, le principe de base est toujours le même. Comme l'illustre la Figure 2.10, dans la couche utilisateur, l'infrastructure de communication n'est pas explicite. Le concepteur fait appel aux fonctions de lecture et d'écriture appelées aussi "méthodes de commodité" pour communiquer avec les modules de son environnement. Le socket de l'*initiator* crée l'objet transaction et traduit cet appel vers l'une des fonctions des interfaces de transport. On est dans la couche protocole. La couche de transport est représentée par les chemins d'envoi et de retour. L'implémentation de la fonction de transport est faite au niveau du module qui reçoit la transaction. Celui-ci effectue donc la traduction inverse pour décoder la transaction et l'exécuter.

Les interfaces de transport : Les interfaces de transports constituent la partie normative du standard SystemC-TLM-2 et assurent l'interopérabilité des modèles. Quatre interfaces sont définies : bloquante, non bloquante, DMI⁹ et Debug. Les deux dernières interfaces servent essentiellement à mettre en œuvre un outillage de debug. L'interface DMI permet un accès direct en mémoire en évitant tous les supports de communication (socket, canaux). L'interface Debug est conçue pour faire des transferts sans attentes temporelles et sans effets de bord sur la simulation. La fonction n'effectue aucune véritable opération mais sert uniquement de diagnostic. Ces deux interfaces ne seront pas détaillées dans ce manuscrit. Les deux premières interfaces ont été créées pour supporter deux niveaux de détails temporels. Pour un maximum d'interopérabilité, l'interface de transport utilisée doit grouper au moins les méthodes des interfaces de transport bloquant et non bloquant sans avoir l'obligation d'implémenter les deux.

L'interface de transport bloquant : La force de cette méthode de transport est qu'elle offre un style de codage simplifié et plus abstrait pour les modèles permettant

⁹Direct Memory Interface

d'effectuer une transaction complète par un seul appel de fonction. Le transport se fait alors dans un seul sens : le sens de l'envoi (forward path). La transaction reçue par le *target* est traitée et retournée. Le processus de l'*initiator* reste bloqué en attente de ce retour. Cette fonction fournit ainsi deux détails temporels qui sont le début et la fin de la transaction. L'interface de transport bloquant est définie dans la TLM-2 comme suit :

```
template<typename TRANS=tlm_generic_payload>
class tlm_blocking_transport_if: public virtual public sc_interface
{
    public:
        virtual void b_transport(TRANS& trans,sc_core::sc_time& t) =0;
};
```

PROGRAMME 2.2 – L'interface de transport bloquant

Comme le montre l'extrait de la bibliothèque donné dans le programme 2.2, la méthode de transport bloquant prend en premier argument une référence vers l'objet transaction (de type par défaut `tlm_generic_payload`) et une variable temporelle en deuxième argument. Cette dernière sert à indiquer le point de début et de fin de la transaction relatifs aux temps de simulation courant. Le temps de début correspond au moment de l'appel de cette fonction. Le temps de fin est celui du retour de la fonction. La méthode de transport bloquant "`b_transport`" peut s'exécuter dans un temps de simulation null. Cependant, elle peut contenir un appel à la fonction "`wait(t)`" qui va différer le retour de la fonction à un autre instant de la simulation. Pendant ce temps, l'*initiator* reste bloqué en attente du retour de l'objet transaction.

Les interfaces de transport non bloquant : Contrairement au transport bloquant, le transport non bloquant est composé de deux chemins : le chemin de l'envoi utilisé par l'*initiator* pour envoyer la requête au *target* et le chemin de retour (*backward path*) utilisé par le *target* pour l'envoi de la réponse d'où l'intérêt du port *target* dans l'*initiator_socket*. L'avantage de cette interface est que l'*initiator* peut continuer son exécution après l'envoi de la requête.

L'interface expose deux méthodes de transport permettant une modélisation plus détaillée des interactions entre l'*initiator* et le *target* au cours d'une transaction. La transaction est découpée en plusieurs séquences ou phases dont chacune marque un point temporel dans l'évolution de la transaction. L'ensemble des phases correspond à l'implémentation abstraite d'un protocole de communication. Par défaut, cette interface permet d'implémenter le protocole de base de TLM appelé "*base protocol*" composé de quatre phases correspondant au début et à la fin de la requête et de la réponse. Le mode de transport non bloquant est particulièrement adapté à la modélisation des transferts pipelinés, où un composant peut émettre plusieurs requêtes successives sans attendre les résultats des requêtes précédentes. Cependant, contrairement au transport bloquant, plusieurs détails temporels de la communication peuvent être considérés. Par conséquent, le temps de sim-

ulation avance à chaque fois résultant généralement en une simulation plus lente.

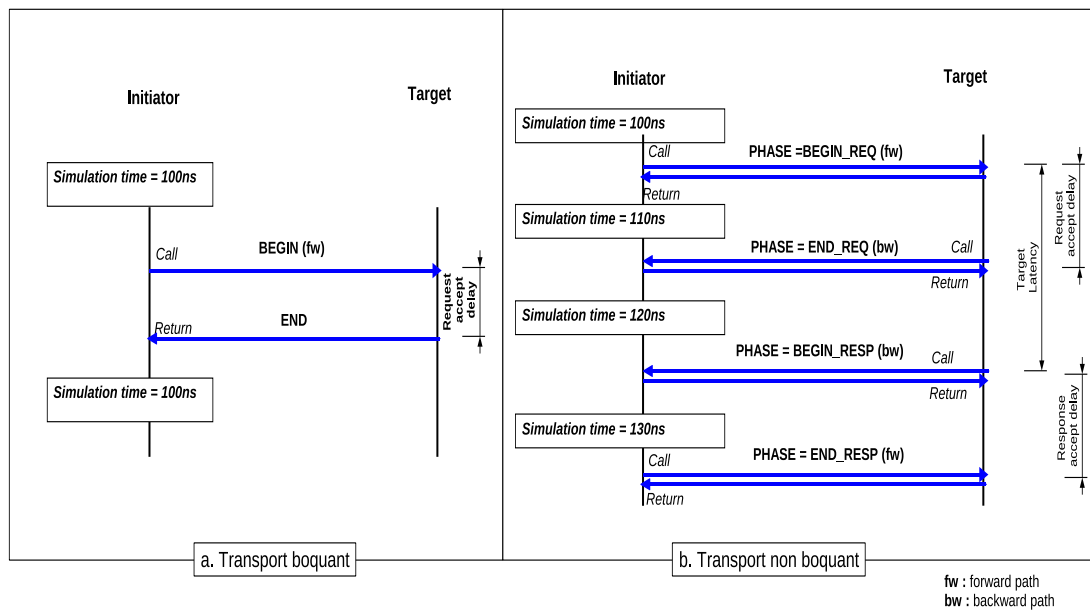


FIGURE 2.11 – Les phases des transports bloquants et non bloquants

La Figure 2.11 illustre les diagrammes de séquence relatifs à un transport bloquant (a) et non bloquant (b). Alors que l'utilisation de l'interface de transport bloquant permet de bloquer le temps de simulation en attente de la fin de la transaction atomique, la communication non bloquante offre une meilleure lisibilité des principales phases du protocole de communication implémenté et une prise en considération des latences de transmission et du temps de traitement des transactions. Chaque phase représente un point de synchronisation entre le processus de l'*initiator* et du *target*.

2.2.1.3 Exemple illustratif : le pv_dma

Nous allons illustrer les notions introduites dans cette section par un exemple extrait de la distribution de la première version d'évaluation de la bibliothèque TLM-2 (TLM-2-Draft¹⁰). Cette version comporte quelques différences par rapport au standard 2.0 mais l'exemple est suffisamment simple pour donner une bonne intuition de la différence de niveau d'abstraction par rapport à RTL. Les caractéristiques de cet exemple sont les suivantes :

- la transaction est décomposée en deux objets :
une requête du type `t1m_request <ADDRESS,DATA>` et une réponse du type `t1m_response <DATA>`.

¹⁰http://www-ti.informatik.uni-tuebingen.de/~systemc/Documents/Presentation-15-OSCI2_aynsley.pdf

- le transport est assuré par une fonction de transport bloquante bi-directionnelle décrite par le programme 2.3 permettant d'envoyer une requête et de recevoir la réponse.
- à la place des *sockets*, on parle de ports. Au niveau de l'*initiator* (respectivement pour le *target*), la classe `tlm_initiator_port` est une extension de la classe `sc_port` (respectivement `sc_export`) de SystemC.

```
//-----
// bidirectional blocking interfaces of pv_core_ifs
//-----
template < typename REQ , typename RSP >
class tlm_transport_if : public virtual sc_interface
{
public:
    virtual RSP transport( const REQ & ) = 0;

    virtual void transport( const REQ &req , RSP &rsp ){
        rsp = transport( req );
    }
};
```

PROGRAMME 2.3 – L'interface de transport de TLM2-D1

La Figure 2.12 est représentative d'un modèle couramment utilisé dans les systèmes sur puce. Le modèle est composé de quatre modules (un micro-processeur CPU¹¹, un DMA¹², et deux mémoires) et d'un canal hiérarchique appelé "ROUTER" sur la Figure 2.12. Le canal est un bus de type *memory-mapped* : il possède un fichier de configuration indiquant les plages d'adresses réservées pour chaque *target* (*memory1*, *memory2* et *DMA*) permettant ainsi au décodeur d'adresses d'orienter la requête reçue vers la bonne destination. Le scénario d'exécution est le suivant : le CPU commence par programmer le DMA pour effectuer des transferts entre les mémoires (lectures suivies d'écritures). Il lui transmet alors les adresses source et destination et la taille du transfert. Lors de la configuration, le DMA joue le rôle de *target* par rapport au CPU. À la fin du transfert demandé par le CPU, le DMA envoie une interruption matérielle (IRQ) vers ce dernier pour lui indiquer qu'il est à nouveau disponible. Pour les transferts mémoire, le DMA joue le rôle de l'*initiator*. Le routeur est un module d'interconnexion. Il possède un export pour la réception des transactions des modules *initiator* et un port pour leur acheminement vers les destinations finales.

Selon la légende de cette figure, on peut distinguer deux types de transferts :

- une liaison directe entre le DMA et le CPU faite par le signal booléen d'interruption

¹¹Central Processing Unit

¹²Direct Memory Access, pour contrôleur d'accès direct à la mémoire. C'est un composant permettant d'accéder directement à la mémoire vive sans occuper de ressources processeur.

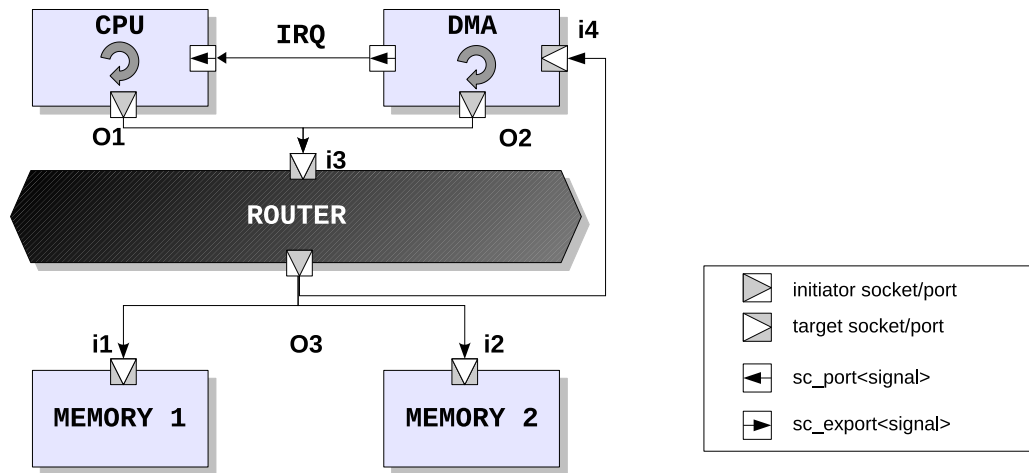


FIGURE 2.12 – Les modèles d'abstraction SystemC du DMA

IRQ. Cette liaison n'utilise pas les interfaces de communication de TLM mais elle est conforme à une liaison SystemC du type port-export (*c.f.* section 2.2.1.1).

- des transferts qui passent à travers le support de communication qui utilisent les interfaces de transport bloquant de TLM.

On s'intéresse particulièrement à la description des transferts transactionnels bloquants. Les ports *initiator* o1, o2, os3 sont du type `pv_initiator_port<ADDR,DATA,N>`. Comme le montre l'extrait de code du programme 2.4, cette classe est une extension du port de TLM-2D1 qui permet de définir les fonctions de commodité de la couche application : `read`, `read_block`, `write`, `write_block`. Ainsi, afin de démarrer une transaction, le module doit faire appel aux fonctions de commodité de son port. Ces fonctions utilisent les paramètres qui lui sont passés pour appeler à leur tour la fonction de transport. Dans l'extrait donné, seule la fonction de lecture simple est détaillée, à titre indicatif. Les méthodes `read_block` et `write_block` permettent respectivement de lire et écrire un bloc de données de taille paramétrable.

```
//-----
// pv_initiator_port class
//-----
//ADDRESS and DATA have int type
template<typename ADDRESS,typename DATA,int N = 1>
class pv_initiator_port :
    public tlm::tlm_initiator_port<
        tlm::tlm_transport_if
        <tlm::tlm_request<ADDRESS,DATA>,
        tlm::tlm_response<DATA> >, N>,
        ...
    public virtual pv_if<ADDRESS,DATA>, //interface for convenience
    methods
```

```

....
//convenience method
virtual tlm::tlm_status read(const ADDRESS& address,
                           DATA& data,
                           const unsigned int byte_enable = tlm::
                               NO_BE,
                           const tlm::tlm_mode mode = tlm::REGULAR,
                           const unsigned int export_id = 0
                           ){
    // create the request and response objects
    tlm::tlm_response<DATA> response;
    tlm::tlm_request<ADDRESS,DATA> request;
    // configure the request
    request.set_command(tlm::READ);
    request.set_address(address);
    request.set_data_array(1,&data); --1 word
    request.set_byte_enable_array(1,&byte_enable);
    request.set_mode(mode);
    // Add the registered target port id to the request
    request.set_tlm_export_id(
        this->get_target_port_list()[target_port_rank]
        ->get_tlm_export_id());
    // call the transport interface
    response = (*this)[target_port_rank]->transport(request);
}
//other convenience methods (write, read_block, write_block)
virtual tlm::tlm_status write(const ADDRESS& address,
                              const DATA& data,
                              const unsigned int byte_enable = tlm::
                                  NO_BE,
                              const tlm::tlm_mode mode = tlm::REGULAR,
                              const unsigned int export_id = 0
                              ){...}
...
}

```

PROGRAMME 2.4 – Le port initiator et les fonctions de commodité

Le code décrit plus précisément un multi-port pouvant être lié à plusieurs `sc_export` dont chacun est associé à un identifiant “`target_port_rank`”. Par défaut, chaque port possède un seul `sc_export` (`target_port_rank=0`); c’est le cas par exemple pour tous les composants du modèle à l’exception du routeur. Dans le cas du routeur, une fonction de décodage d’adresse basée sur un fichier contenant les plages d’adresses de chaque *target* est nécessaire afin de déterminer le `sc_port` vers lequel l’appel à la fonction de transport va être orientée. Le routeur est alors un bus *memory-mapped*. L’interface de transport exposant la fonction de transport bloquant “`transport`” doit être implémentée par tous

les *targets*.

On propose de suivre le scénario d'une opération de lecture faite par le DMA dans la mémoire 1. L'extrait de code donné dans le programme 2.5 contient le constructeur du module DMA. Il faut souligner que le niveau de granularité structurel de l'interface du module TLM n'est pas au niveau *pin-accurate* contrairement au niveau d'abstraction RTL illustré dans la section 2.1. Tous les ports de communication sont encapsulés dans deux structures de données : l'`initiator_port` pour l'envoi des requêtes et le `target_port` pour la réception.

```
//-----
// Constructor
//-----
pv_dma::pv_dma(sc_module_name module_name,
               tlm::tlm_endianness endian) :
    sc_module(module_name),
    pv_slave_base<int,int>(name()),
    //target and initiator port initialisation
    target_port("target_port"),
    initiator_port("initiator_port"), ...
{
    target_port(*this);
    //Process declaration using SC_THREAD macro
    SC_THREAD(transfer);
    //activation event (sc_event m_start_transfer;)
    sensitive << m_start_transfer;
    ...
}
```

PROGRAMME 2.5 – Le constructeur du DMA

Dans le constructeur du DMA, la fonction “`transfer`” est déclarée comme étant un processus SystemC de type “`SC_THREAD`” sensible à l'événement “`m_start_transfer`”. Cette sensibilité est définie via l'instruction “`sensitive << m_start_transfer`”. Contrairement au processus RTL (*c.f.* programme 2.1), le processus TLM n'a pas d'horloge de synchronisation. Son exécution est lancée suite à la notification de l'événement `m_start_transfer`. Selon le scénario d'exécution du code de la plateforme, la notification de cet événement a lieu suite à la réception de l'ordre de début de transfert émis par le CPU (écriture de la valeur “START” dans le registre de contrôle du DMA “`m_pv_dma_control`”).

La portion du code suivant la directive “`#else`” du programme 2.6, est un extrait du code du processus `transfer()` qui montre une succession d'opérations de lecture/écriture élémentaires permettant de transférer un bloc de données de taille `m_pv_dma_length` de la source vers la destination. Ces transferts sont initiés par le DMA via son *initiator port*

o2. L'ensemble des variables et des constantes référencées dans ce code est défini dans le fichier "pv_dma.h" associé.

```
//-----
// dma transfer management
//-----
void pv_dma::transfer() {
    tlm::tlm_status status; //response status
    while(1) {              //DMA process
        if (m_pv_dma_control & START) {
            // starts the memory transfer
#ifdef PV_DMA_BLOCK_TRANSFER
            int nb_word = m_pv_dma_length/tlm::get_nb_byte<int>();
            int tmp[nb_word];
            status = initiator_port.read_block(m_pv_dma_src_addr,tmp,nb_word);
            ...
            status = initiator_port.write_block(m_pv_dma_dst_addr,tmp,nb_word);
            ...
#else
            int tmp;
            //tlm::get_nb_byte<int>()=4
            for(int i = 0;i<m_pv_dma_length;i+=tlm::get_nb_byte<int>()) {
                // call the convenience method "read" on the initiator port
                status = initiator_port.read(m_pv_dma_src_addr + i,tmp);
                ...
                //write data tmp to m_pv_dma_dst_addr+i
                status = initiator_port.write(m_pv_dma_dst_addr + i,tmp);
                ...
            }
#endif
        }
    }
}
}
....
```

PROGRAMME 2.6 – Le processus de lecture-écriture dans les mémoires

L'*initiator port* du DMA est relié au *target port* du routeur. Les appels de fonctions de lecture/écriture seront alors dirigés vers ce composant. Le routeur a pour charge de recevoir la requête, d'extraire l'adresse destination afin de la décoder et de rediriger la requête vers sa destination finale via le port o3. Dans le cas du DMA, les cibles sont les mémoires. Celles-ci doivent alors fournir une implémentation de la fonction de transport et des fonctions de commodité : la requête est décodée pour en extraire l'opération demandée (lecture, écriture) et les fonctions de commodité sont appelées par la suite pour exécuter la commande et retourner la réponse demandée. Le scénario complet est schématisé par le diagramme de séquence de la Figure 2.13 :

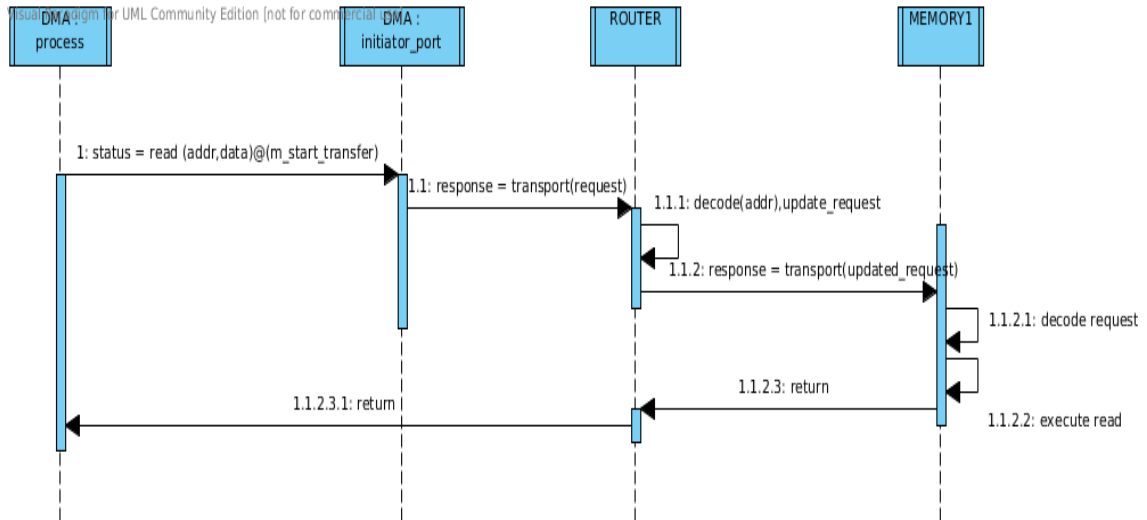


FIGURE 2.13 – Le diagramme de séquence des appels de fonctions pour faire une lecture en mémoire par le DMA

La succession des opérations de lecture\écriture peut être remplacée par une seule opération permettant d'effectuer en un seul transfert atomique la lecture ou d'écriture d'un bloc de données de taille `m_pv_dma_length`. La portion du Programme 2.6 délimitée par les directives “`#ifdef PV_DMA_BLOCK_TRANSFER`” et “`#else`” montre les opérations permettant d'effectuer ces transferts. La fonction `write_block`, par exemple, est l'équivalent d'une succession de `nb_word` appels à la fonction `write` décrite dans le programme 2.6. Ainsi, le niveau de granularité des données est différent entre TLM et RTL. Contrairement aux transferts RTL, les fonctions de communication TLM permettent la transmission d'un bloc de données en un seul appel de fonction. Ce bloc peut être une image, un fichier, etc.

2.2.2 Les niveaux d'abstraction

Le cœur du flot ESL est la plateforme virtuelle. Comme le simple cas du modèle décrit dans la Figure 2.12, une plateforme virtuelle est une représentation fonctionnellement précise du système matériel, capable de s'exécuter sur un poste de travail. Un de ses principaux objectifs est de permettre de démarrer au plus tôt le développement et la validation du logiciel. Par conséquent, cette description doit contenir suffisamment de détails afin de permettre l'exécution du logiciel, des drivers, des i-OS¹³ et des applications dans un temps relativement court. Dans la branche matérielle du flot de conception, cette plateforme permet l'exploration de l'architecture et facilite la prise de décisions dès les premiers niveaux de modélisation.

Dans cette partie, on va identifier les différents niveaux de précision possibles grâce à

¹³integrated Operating system

l'utilisation de SystemC-TLM.

2.2.2.1 Les niveaux d'abstraction Transactionnels

Le principe de séparation des aspects de calcul et de communication implémenté dans les bibliothèques SystemC et TLM permet de distinguer deux types de composants : d'une part les composants qui assurent les communications (les canaux et les ports) et de l'autre part les composants qui se chargent des calculs (les PEs¹⁴).

Dans TLM, le raffinement intéresse particulièrement les communications. Les différents niveaux d'abstraction transactionnels sont précis au niveau fonctionnel (*functionally accurate*). La distinction entre niveaux d'abstraction transactionnels se fait principalement par rapport au niveau de granularité du temps.

Le standard SystemC-TLM-2 ne donne pas une définition restrictive du niveau d'abstraction qui doit être utilisé pour chaque cas d'utilisation (développement logiciel, évaluation des performances, vérification du matériel, etc.) mais il fait la distinction entre les styles de codages, les interfaces de transport et les cas d'utilisation. La Figure 2.14 montre que les interfaces et les différents concepts de communication présents dans la couche intéropérabilité du standard sont les mécanismes de base pour la description du modèle transactionnel de référence. Les styles de codages sont des directives d'utilisation permettant une utilisation optimale de la bibliothèque.

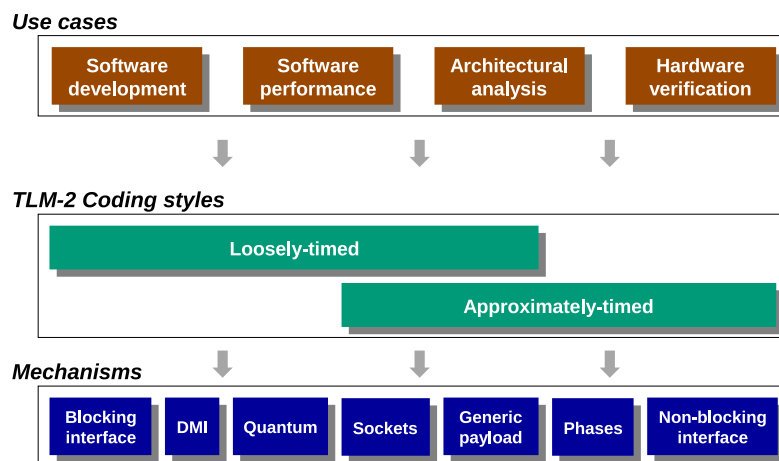


FIGURE 2.14 – Les cas d'utilisation, les styles de codage et les mécanismes de programmation (source :[OSC09])

Le modèle non-temporisé est une spécification fonctionnelle exécutable du système contenant toutes les fonctionnalités demandées en faisant abstraction des détails temporels. La communication entre les modules est faite implicitement par des appels de fonctions sans avoir besoin nécessairement de modéliser les structures de communication.

¹⁴Processing Elements

Dans la plupart des cas, ce style de modélisation n'est pas considéré comme étant transactionnel car ce dernier nécessite des interactions entre plusieurs processus concurrents avec un certain degré de précision temporelle permettant l'exécution du logiciel embarqué. TLM-2 propose deux styles de codage :

- Loosely timed (LT)-PV dans TLM1- : Les modèles de ce niveau contiennent une quantité limitée de détails temporels dans le but d'avoir une simulation plus rapide(1.000-10.000x)[BM10] et un temps de développement plus court (10-100x)[BM10] comparé à celui de RTL. Le niveau de détail requis doit permettre d'identifier individuellement les transactions. Pour cette raison, l'interface de transport bloquant est plus appropriée à ce niveau d'abstraction étant donné que la transaction est achevée dans un seul appel de fonction.
- Approximately timed (AT)-PVT dans TLM1- : Les modèles de ce niveau contiennent suffisamment de détails temporels pour permettre l'exploration de l'architecture, l'évaluation des performances et le développement du logiciel temps réel. Les interfaces de transport non bloquant sont plus adaptées à ce style de codage étant donné qu'elles permettent de distinguer plusieurs détails temporels relatifs à une même transaction (au moins quatre). La simulation est par conséquent plus lente mais le modèle est plus précis.

Le niveau d'abstraction des communications *cycle-accurate*, à son tour, ne fait pas partie des niveaux d'abstraction définis dans le standard TLM-2. Cependant, il est possible de créer des modèles à ce niveau en utilisant SystemC et éventuellement la TLM-1. Bien que ce cas de figure n'est pas explicitement décrit dans le standard actuel, un modèle AT peut être raffiné en un modèle de communication *cycle-accurate* par l'inclusion de détails temporels supplémentaires permettant de décrire le déroulement du protocole de communication à chaque cycle de l'horloge du bus. Il faut noter que le niveau de précision peut varier dans une même catégorie selon le type de l'application et du logiciel à embarquer. Par exemple, le développement de logiciels d'application ne nécessite quasiment pas de détails temporels. Par contre, si la plateforme virtuelle est destinée au déploiement d'un système d'exploitation alors le modèle doit mimer certains aspects structurels de l'implémentation. Dans le cas où le logiciel destiné à s'exécuter sur la plateforme virtuelle est un logiciel de traitement de données, tel qu'une application d'encodage audio/vidéo, alors la précision temporelle doit être plus importante (*cycle-accurate*) pour permettre l'exécution du logiciel en temps réel.

La Figure 2.15 illustre les différents niveaux d'abstraction d'une plateforme virtuelle. Cette figure permet d'établir un lien entre les niveaux d'abstraction définis dans la TLM-2, la TLM-1 et ceux couramment utilisés en industrie tels qu'on a présenté dans 1.2.2.

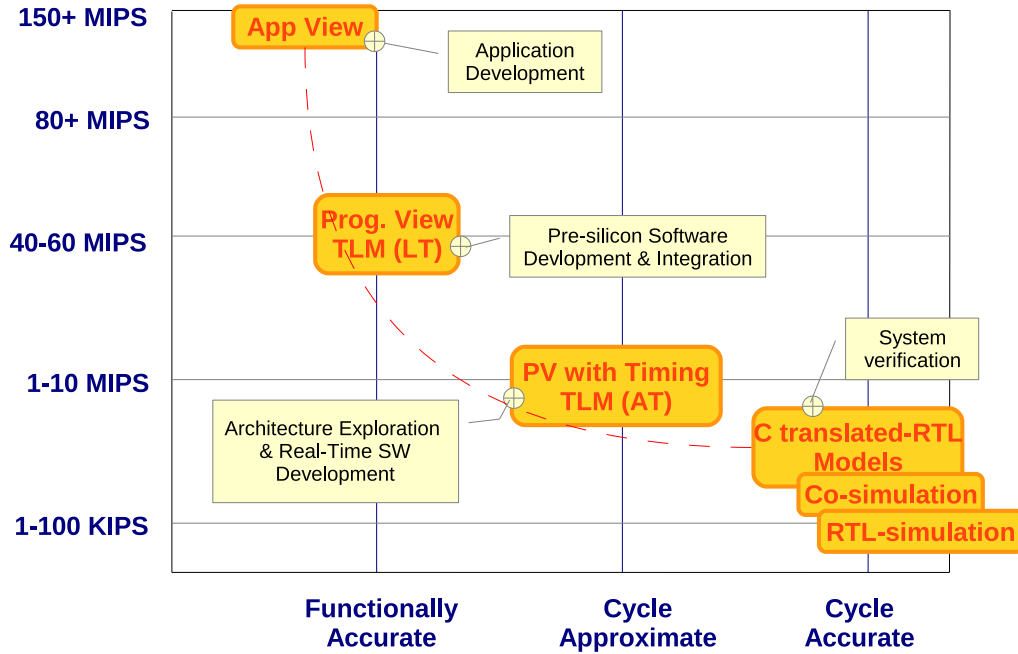


FIGURE 2.15 – Les niveaux d'abstraction et cas d'utilisation (source :[Sch09])

2.2.2.2 Le flot ESL détaillé

Grâce au principe de séparation des profils, l'injection de détails temporels ou architecturaux peut toucher l'aspect de communication indépendamment de l'aspect de calcul et vice-versa. Dans [CG03] ceci est décrit comme un graphe de modélisation à deux axes où le modèle peut évoluer selon l'un ou les deux axes. Il faut noter par ailleurs qu'à la date de publication de cet article, la bibliothèque TLM2 n'avait pas encore été définie.

En se basant sur un croisement du graphe de modélisation de Gasjki et al. avec la TLM-2, et en se référant à ce qui a été proposé dans [BM10][Wal05][Sch09], on peut déduire un résumé des différents niveaux d'abstraction. La Figure 2.16 représente un flot descendant allant de la spécification à l'implémentation.

Plusieurs outils de conception et de synthèse alimentent le passage d'un niveau à un autre dans le flot descendant. Les outils de synthèse de haut niveau (*HLS :High Level Synthesis*) permettent par exemple de générer l'équivalent RTL d'un modèle algorithmique de composants calculatoires, de traitement du signal par exemple, en prenant en considération les contraintes de temps, de consommation et d'espace. Nous citons par exemple, l'outil Catapult de MentorGraphics¹⁵, C-to-Silicon de Cadence¹⁶ et Symphony de Synopsys¹⁷. Parmi les solutions académiques, nous citons l'outil GAUT¹⁸[CAC⁺07] dédié à la synthèse haut niveau des applications de traitement de signal numérique (DSP).

¹⁵acqui en 2011 par Calypto, <http://calypto.com/en/products/catapult/overview/>

¹⁶[http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx?](http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx?CMP=011414_ctosilicon_pr)

¹⁷<http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/default.aspx>

¹⁸<http://www.gaut.fr/>

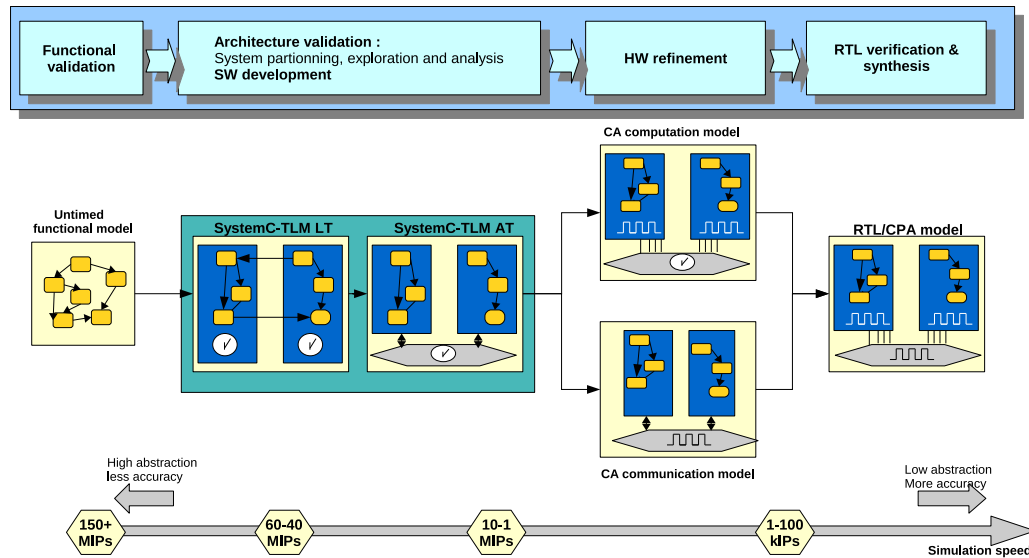


FIGURE 2.16 – Les modèles d'abstraction en SystemC-TLM

C'est un logiciel libre utilisé notamment dans le milieu universitaire. Comme l'illustre la Figure 2.17, l'outil prend en entrée un programme C et un ensemble de contraintes de synthèse données par l'utilisateur et génère en sortie une description en VHDL synthétisable décrivant une architecture composée potentiellement d'une unité de calcul, d'une mémoire et d'une unité de communication et de multiplexage.

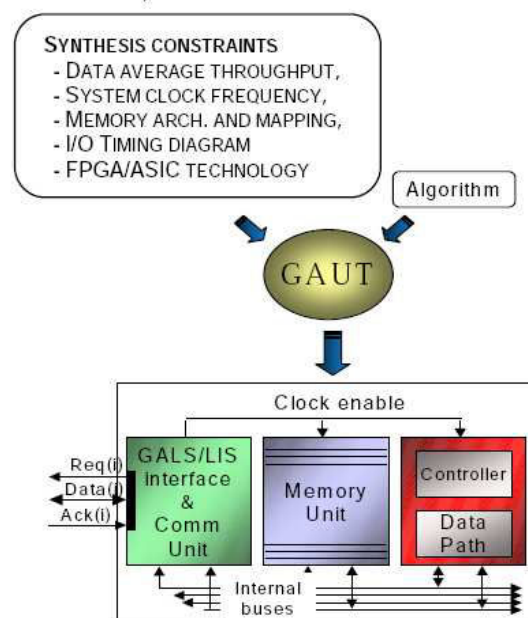


FIGURE 2.17 – L'outil HLS GAUT (source :[Bou07])

Le processus de transformation nécessite dans la plupart des cas l'intervention de l'utilisateur pour faire des choix et fournir des détails concernant notamment des informations spécifiques à l'implémentation. Le raffinement d'une plateforme se fait généralement unité

par unité[Man13].

Le flot représenté est donné à titre indicatif. Contrairement aux méthodologies de conception qui sont quasiment stables à partir du niveau RTL, les techniques et les langages de modélisation de haut niveau sont en permanente évolution. Pour cette raison, nous considérons que les deux niveaux stables sont le niveau transactionnel et le niveau de transfert des registres. De plus, le processus de raffinement peut aussi changer selon le type du composant (calcul, communication). En résultat, des composants à des niveaux d'abstraction différents peuvent co-exister dans le même modèle résultant en un modèle hybride

2.3 Le raffinement des modèles

Le terme raffinement décrit une transition d'un niveau abstraction vers un autre niveau moins abstrait. Cette transition implique l'ajout d'informations spécifiques au niveau d'abstraction cible et la suppression des informations devenues obsolètes. Elle assure la continuité du processus de conception.

La transition majeure dans le flot de conception ESL consiste à passer du niveau transactionnel au niveau RTL. Nous rappelons qu'aux niveaux transactionnels, les interfaces de communication offrent un ensemble de fonctions qui fournissent une vue abstraite du protocole de communication. Le passage au niveau RTL consiste à remplacer les composants par leurs équivalents RTL. Ceci implique la modification des structures de données traitées et des interfaces des composants. Au niveau RTL, la "transaction" est remplacée par un ensemble de signaux binaires qui reflètent l'interface d'un protocole de communication bien déterminé (wishbone, AHB, etc.). La synchronisation des communications et des traitements est faite par une horloge. L'évolution de ce protocole est définie pour chaque cycle de l'horloge de synchronisation. Les types de données utilisés doivent être binaires (*bit-accurate*) et synthétisables. Les interfaces doivent fournir des ports destinés à être connectés aux signaux transportant les données traitées et échangées avec l'environnement du composant (*pin-accurate*). Ainsi, tous les ports du protocole de communication de bas niveau doivent être représentés.

Comme nous l'avons mentionné, le flot de développement n'est pas parfaitement linéaire. Dans la plupart des cas, des composants particuliers dits "transacteurs" permettent de lier les composants appartenant à des niveaux d'abstraction différents. Un transacteur peut avoir plusieurs significations mais dans le sens général, on peut le définir comme un adaptateur entre deux domaines ou niveaux d'abstraction. Quand son rôle consiste à assurer le passage d'un domaine plus abstrait vers un domaine moins abstrait, il peut alors être nommé *driver*[Men06]. Dans le sens inverse, l'objet miroir peut être appelé *responder*[Men06]. Selon son rôle dans la communication, le module peut être relié à l'un

ou aux deux types de transacteurs. En plus de leur utilisation dans la modélisation des systèmes hybrides, les transacteurs peuvent aussi être utilisés pour la vérification dynamique basée sur les transactions [Swa06][BCG⁺00] ou pour l'émulation des systèmes transactionnels (co-émulation)[Kul08]. Dans le premier cas, ils sont souvent référencés comme étant un *bus functional model (BFM)*[BCG⁺00]. Les auteurs de [RN03] décrivent l'utilisation des transacteurs afin de valider le modèle RTL en comparant ses sorties à celles du modèle de référence en réponse aux mêmes stimuli comme l'illustre la Figure 2.18.

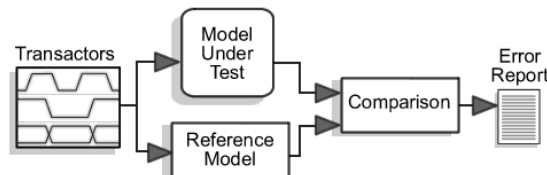


FIGURE 2.18 – La vérification du DUT par rapport au modèle de référence[RN03]

La première version du standard TLM proposait déjà des exemples de plateformes hybrides incluant des transacteurs permettant la conversion des communications entre des composants TLM et RTL[RSPF05]. Un transacteur possède deux interfaces dont chacune correspond à l'un des deux domaines et implémente un processus synchrone lui permettant de faire cette conversion.

La Figure 2.19 décrit un transacteur pour un bus wishbone. Ce transacteur permettra par exemple, de connecter un CPU codé au niveau TLM au routeur RTL.

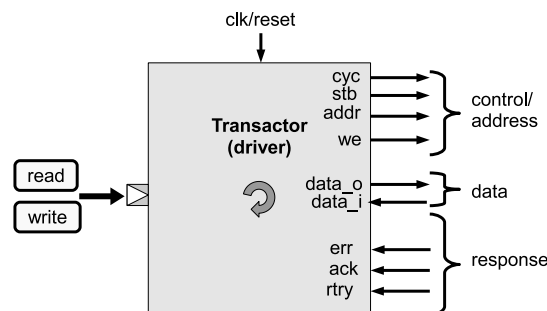


FIGURE 2.19 – Un transacteur TLM-2 vers wishbone

Il faut noter que de tels composants ne fournissent toutefois pas un moyen sémantique formel pour raisonner sur le raffinement de propriétés logico-temporelles.

2.4 Bilan

Par définition, la modélisation est une technique qui consiste à donner une représentation structurée et abstraite afin d'améliorer la compréhension du problème et de simplifier son traitement. C'est la raison pour laquelle les industriels s'orientent de plus en plus vers

des flots de conception impliquant des niveaux d'abstraction plus élevé que RTL : ESL. Dans ce cadre, on s'intéresse particulièrement au flot de conception ESL descendant basé sur SystemC-TLM largement adopté par les industriels. Selon ce paradigme, la conception commence par une spécification exécutable au niveau système qui sera raffinée de manière incrémentale pour atteindre le niveau RTL en passant par le niveau transactionnel.

L'objectif du passage du niveau RTL au niveau système ne se limite pas uniquement au besoin de simplification des descriptions des systèmes complexes mais il témoigne d'une nouvelle philosophie : "concevoir et vérifier" au lieu de "concevoir ensuite vérifier". En effet, l'adoption du flot ESL permet de démarrer le développement du logiciel dès les premières étapes de conception et principalement de vérifier les fonctionnalités du modèle de référence avant de s'engager dans le processus de raffinement et introduire des détails supplémentaires. Cette étape est primordiale et la qualité du système est étroitement liée à la qualité de la vérification.

Dans cette thèse nous nous concentrons sur la vérification à base d'assertions (ABV) détaillée dans le chapitre suivant.

Contexte : La vérification à base d'assertions

Sommaire

Introduction	86
4.1 Les motivations et enjeux du raffinement	86
4.1.1 Les différences structurelles	86
4.1.2 Les différences temporelles	87
4.1.3 L'exemple du pv_dma	90
4.2 Travaux sur le raffinement des assertions	91
4.2.1 Approches à base de transacteurs	91
4.2.2 Approche de Ecker et al.	92
4.2.2.1 La transformation des assertions	93
4.2.2.2 Les restrictions	94
4.3 Solution de raffinement	95
4.3.1 Premières règles de raffinement	95
4.3.1.1 Règles de transformation impliquant l'opérateur <code>before</code>	95
4.3.1.2 Règles de transformation impliquant l'opérateur <code>until</code>	101
4.3.1.3 Règles de transformation impliquant l'opérateur d'im-	
plication logique	102
4.3.2 Alternative : utilisation des SEREs	104
4.4 Bilan	105

“Everyone knows that debugging is twice as harder as writing a program in the first place”.

Brian Kerninghan -Elements of Programming style-1974

Introduction

Face à des utilisateurs de plus en plus exigeants, les concepteurs sont amenés à développer des SoCs de plus en plus complexes contenant plusieurs composants matériels et logiciels pour la réalisation de fonctions de plus en plus sophistiquées dans des délais de plus en plus réduits. Aujourd’hui, 57% des SoCs ont au moins deux processeurs embarqués[Fos13]. La qualité est un critère déterminant dans la réussite ou l’échec d’un produit. Pour les SoCs complexes, garantir un niveau de qualité acceptable à des prix compétitifs est devenu un défi de taille. Malgré l’augmentation de 75% du nombre moyen d’ingénieurs chargés de la vérification entre 2007 et 2013[Fos13] pour tenter d’absorber la charge de vérification, celle-ci reste toujours l’activité la plus contraignante dans le cycle de développement. Pour limiter le temps de vérification sans affecter la qualité de leurs produits, les industriels font appel à plusieurs techniques de vérification statiques, dynamiques et semi-formelles (*c.f.* section 1.2.4.1). Nous nous intéressons à une technique particulière dite vérification basée sur les assertions (ABV) pour la vérification fonctionnelle des SoCs au cours de la simulation. L’utilisation de l’ABV permet globalement de réduire de 50% le temps de vérification[ABG⁺00][TS08] et contribue par conséquent à l’amélioration de la qualité des SoCs. Dans ce chapitre, nous allons présenter l’application de l’ABV aux niveaux RTL et TLM en utilisant un langage standard pour la spécification des assertions appelé PSL (*Property Specification Language*)[PSL05, PSL10]. Un passage par ce langage s’impose.

3.1 La spécification à base d’assertions :PSL

Deux langages de spécification formelle des propriétés logico-temporelles sont définis : SVA, associé à SystemVerilog, et PSL (Property Specification Language). Ce sont deux standards IEEE qui permettent une spécification non ambiguë, compacte et commode des comportements souhaités. PSL est un langage déclaratif de spécification formelle des propriétés logico-temporelles basé sur le langage Sugar2.0¹ défini depuis 1994 par IBM Haifa³. L’objectif de Sugar2 était de réduire la réticence des industriels face aux techniques de l’ABV en facilitant l’utilisation de la logique temporelle CTL⁴[HR04] et

¹Sugar est un formalisme pour la spécification des comportements temporels bâti sur la sémantique² de LTL et d’une extension syntaxique du langage CTL

³<https://www.research.ibm.com/haifa/projects/verification/sugar/index.html>

⁴Computation Tree Logic

LTL⁵[HR04]. PSL est issu de l'initiative du comité technique de la vérification formelle (FVTC⁶) de Accellera réunissant plusieurs industriels œuvrant pour la définition d'un langage mathématiquement correct, automatisable et facile à interpréter et à écrire. Il a été reconnu comme un standard Accellera depuis 2003 avant d'être un standard IEEE Std 1850-2005 depuis 2005. Aujourd'hui on est à la deuxième version Std 1850-2010.

Dans le standard PSL, une propriété est définie comme “une expression à base d'opérateurs logiques et temporels entre des expressions booléennes, séquentielles ou entre des sous propriétés dont l'agrégation décrit un ensemble de comportements”[PSL05, PSL10].

Conformément à cette logique, PSL a été construit en quatre couches complémentaires. Cette structure hiérarchique rend possible l'expression de propriétés complexes à partir de propriétés plus simples des couches précédentes :

- couche **booléenne** : elle est formée d'expressions booléennes qui sont évaluées dans un seul cycle d'évaluation. Ces expressions représentent des états ponctuels du système. Cette couche constitue le lien entre l'assertion et le modèle à surveiller. PSL emprunte la syntaxe du langage HDL du modèle à vérifier pour l'expression des conditions booléennes (opérateurs logiques, structures de données, etc.). Les langages supportés sont :VHDL, Verilog, SystemVerilog, SystemC et GDL⁷ et constituent les cinq *flavors* de PSL. Ainsi, la conjonction des deux expressions booléennes a et b , par exemple, est exprimée différemment selon le langage HDL sous-jacent : a **and** b en VHDL, a **&&** b en Verilog et en SystemC.
- couche **temporelle** : elle contient tous les opérateurs temporels permettant de décrire des relations dans le temps entre les expressions de la couche booléenne. Ces opérateurs sont classés en deux catégories : les opérateurs FL, acronyme de *Foundation Language* et OBE pour *Optional Branching Extension*. Les opérateurs de la deuxième classe sont issus des opérateurs définis dans CTL et sont utilisés pour vérifier une propriété sur une structure arborescente modélisant le système. Cette sémantique est plus adaptée à une approche de vérification statique. Nous nous focalisons uniquement sur la vérification dynamique. Cette classe sort alors du cadre de notre travail. Les opérateurs FL seront évoqués par la suite(*c.f.* section 3.1.1).
- couche **vérification** : elle fournit les directives indiquant à l'outil de vérification la manière de traiter la propriété. La directive “*assert*” indique à l'outil que la propriété décrit un comportement à respecter. La directive “*assume*” indique à l'outil qu'il doit considérer la propriété comme une hypothèse sur les entrées.
- couche **modélisation** : cette couche utilise aussi la syntaxe du langage HDL du

⁵Linear-Time Temporal Logic

⁶Formal Verification Technical Committee

⁷General Description Language

modèle pour la déclaration de signaux et de variables auxiliaires. Ces variables peuvent servir, par exemple, à mémoriser un compteur ou pour faire des calculs de données qui n'appartiennent pas forcément au modèle à vérifier.

Une propriété PSL est déclarée dans une structure dite *vunit*. Cette structure peut contenir plusieurs propriétés pouvant partager des variables globales déclarées dans la couche modélisation commune. Un aperçu des différentes composantes de cette structure est illustré par la Figure 3.1.

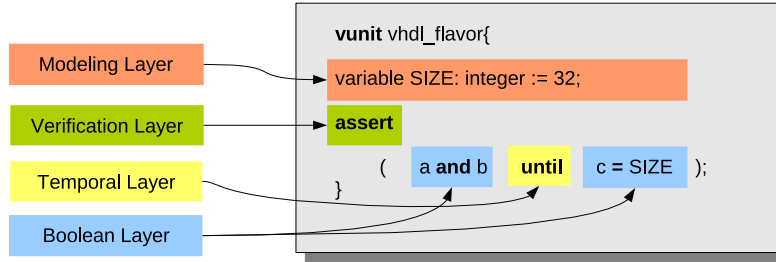


FIGURE 3.1 – Un exemple d'une *vunit* PSL

Les propriétés PSL peuvent être fournies en entrée d'un outil de vérification dynamique (un simulateur, un simulateur) ou statique (un *model checker*, un *theorem prover*). On s'intéresse particulièrement au premier cas. Les moniteurs de surveillance (*checker* - c.f. section 1.2.4.1) sont générés automatiquement dans le même langage que celui du modèle qu'ils surveillent et ils se chargent de notifier le respect ou de la violation de la propriété durant la simulation.

PSL considère une notion discrète du temps : le temps est composé d'un ensemble de cycles d'évaluation. Chaque cycle d'évaluation décrit un état du système. Cet état est représenté par les valeurs des expressions booléennes de la propriété à chaque cycle. La signification d'une propriété est alors définie relativement à cette séquence de cycles appelée aussi "trace" ou "chemin". Les traces seront souvent illustrées par les diagrammes de temps (ou chronogrammes) des expressions de la propriété. Sur certaines traces, nous allons utiliser les lettres "H", "P" et "F" pour dénoter l'état de l'assertion à chaque cycle d'évaluation. Ces lettres correspondent respectivement aux niveaux de satisfaction **H**olds, **P**ending et **F**ail définis plus tard dans la section 3.1.1.3.

3.1.1 Les opérateurs FL

Cette classe regroupe un ensemble d'opérateurs temporels issus d'une part de la sémantique des opérateurs de LTL (*Linear Temporal Logic* pour logique temporelle linéaire) et d'autre part des expressions régulières. Contrairement aux opérateurs OBE, les opérateurs de la logique temporelle linéaire servent à évaluer la propriété sur une seule trace où le temps avance de manière linéaire. Bien qu'il soit possible de l'utiliser dans les approches de vérification statique, cette logique est plus adaptée pour la vérification dynamique.

L'objectif de PSL est de fournir une syntaxe plus intuitive que les opérateurs LTL et CTL symboliques. Par exemple, la réactivité est une propriété couramment vérifiée dans les systèmes concurrents. Dans un système concurrent implémentant le protocole de *hand-shaking*, la spécification de la propriété de vivacité peut être énoncée comme suit : “*toutes les requêtes doivent être suivies par des acquittements*”. Supposons que la variable *req* représente la requête et *ack* représente l'acquiescement. La formule LTL correspondante est $\Box(req \rightarrow \bigcirc ack)$. La propriété PSL équivalente est $P_1 = \text{always}(req \rightarrow \text{next}(ack))$. L'opérateur “**always** (operand)” indique que la propriété doit être évaluée à tous les instants, tout au long de la trace. L'opérateur d'implication logique est “*left_operand* \rightarrow *right_operand*”.

L'opérateur “**next**(operand)” indique que l'opérande doit être satisfait au prochain instant d'évaluation de la trace. Ainsi, si *req* a lieu à l'instant t , alors *ack* doit avoir lieu à l'instant $t + 1$ pour satisfaire la propriété P_1 . Le chronogramme de la Figure 3.2 donne un exemple de trace qui viole la propriété P_1 à l'instant #5. D'autres opérateurs temporels seront étudiés plus loin.

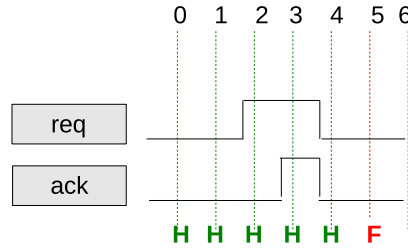
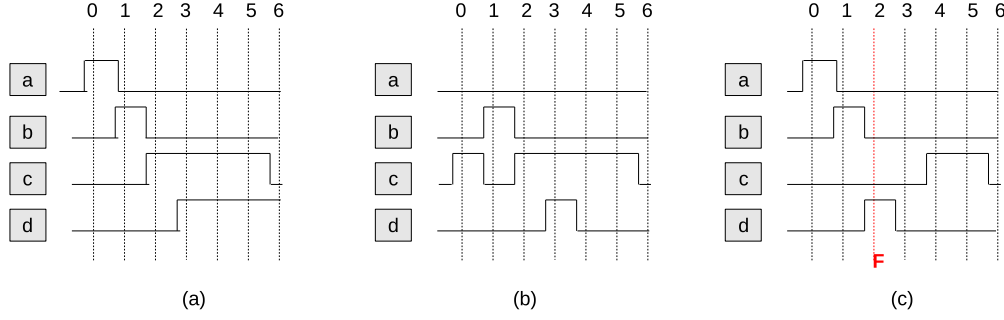


FIGURE 3.2 – Exemples de traces pour la propriété P_1

Le sous ensemble SERE (*Sequential Extended Regular Expressions*) est une extension des expressions régulières classiques utilisées notamment pour le filtrage par motifs. Les opérateurs proposés permettent d'exprimer des propriétés basées sur des séquences de signaux. À la base de cet ensemble se trouvent les expressions régulières classiques : séquentialité $\{;, :\}$, répétitions consécutives $[+i]$, $[*i]$, auxquels s'ajoutent des opérateurs complexes de répétition non consécutives $[\rightarrow i]$, $[= i]$. Ces derniers peuvent être réécrits en termes d'opérateurs classiques. La sémantique de ces opérateurs est très semblable à celle des expressions régulières classiques ; les SERES utilisent des expressions booléennes pour la constitution du comportement séquentiel. La propriété $P_2 = \{a\} \mid \Rightarrow \{b; c[+]; d\}$ est un exemple d'utilisation des SERES dans PSL. Elle signifie que “*si le motif a est reconnu à l'instant d'évaluation présent, alors on devra observer b à l'instant suivant suivi à l'instant d'après d'une répétition non vide de c et à l'instant suivant de d* ”. L'absence de l'opérateur **always** signifie que la propriété doit être évaluée une seule fois, à partir de l'instant #0. Les traces de la Figure 3.3 (a) et (b) satisfont P_2 tandis que celle de la Figure 3.3 (c) la viole.

FIGURE 3.3 – Exemples de traces pour la propriété P_2

Il faut souligner qu'une propriété PSL peut être composée d'opérateurs de la logique temporelle linéaire et de SERES. Dans la suite de ce manuscrit, nous appellerons séquence une expression PSL construite avec des opérateurs SERE et formule, une expression PSL construite à partir d'opérateurs FL.

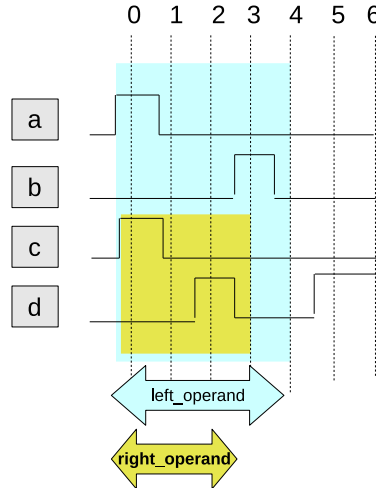
3.1.1.1 Le sous ensemble simple PSL_{ss} (*Simple Subset*)

Dans le sous ensemble simple de PSL, le temps évolue linéairement de droite à gauche à travers la propriété. Cette contrainte permet une évaluation de la propriété au cours de la simulation. Par conséquent, les outils de vérification dynamique sont associés aux propriétés qui appartiennent à ce sous ensemble.

Considérons par exemple, le cas de l'opérateur d'implication logique `left_operand -> right_operand`, la propriété doit demander l'évaluation de l'opérande gauche avant l'évaluation de l'opérande droit. Ainsi, la propriété $P_3 = \text{always } (a \ \&\& \ \text{next}[3]b) \rightarrow (c \ \&\& \ \text{next}[2]d)$ n'appartient pas au sous ensemble simple. Dans la trace illustrée par la Figure 3.4, les évaluations des opérandes gauche et droit sont imbriquées. Le calcul de l'opérande gauche nécessite d'avancer jusqu'au quatrième cycle d'évaluation alors que le calcul de l'opérande droit nécessiterait un retour arrière de trois cycles pour évaluer c .

Afin d'éviter de telles situations, PSL a défini un ensemble de règles syntaxiques qui régissent le sous ensemble simple[PSL05]. Dans le cas de l'opérateur d'implication logique, par exemple, PSL exige que l'opérande gauche soit une expression booléenne afin d'empêcher l'utilisation de tous les opérateurs temporels susceptibles de créer une évolution dans le sens inverse de l'évolution du temps. Bien que la définition soit inchangée, la deuxième version du standard PSL[PSL10] contient une révision des règles syntaxiques du sous ensemble simple.

Les contraintes imposées par PSL sont des contraintes de forme, il est parfois possible de transformer une propriété n'appartenant pas au sous ensemble simple en une autre propriété du sous ensemble en gardant la même signification.

FIGURE 3.4 – Exemples de traces pour la propriété P_3

3.1.1.2 Les opérateurs faibles et forts

PSL propose la plupart de ses opérateurs temporels en deux versions : faible et forte. La version forte vise à s'assurer qu'aucun comportement non souhaité ne puisse avoir lieu jusqu'à atteindre une condition de terminaison. Cette condition doit inévitablement avoir lieu dans la trace d'exécution. La version faible autorise la satisfaction de l'opérateur si la trace ne contient pas la condition de terminaison.

Pour certains opérateurs, la distinction entre version faible et forte n'a aucun sens, c'est le cas par exemple pour les opérateurs **always** et **eventually!**. Le dernier n'existe qu'en version forte. Pour l'opérateur **always**, il n'existe pas de condition de terminaison d'où l'inutilité de la version forte. L'évaluation se fait sur toute la trace d'exécution.

3.1.1.3 Les niveaux de satisfaction des FL

Dans PSL, une propriété doit avoir l'un de ces quatre états au cours de l'évaluation :

- **Satisfaite fortement** (*Holds Strongly*) : Cet état est définitif. La propriété est satisfaite par la trace à l'instant courant et par toute extension de la trace dans le futur car toutes les obligations de la propriété ont été rencontrées.
- **Satisfaite** (*Holds*) : La propriété est satisfaite par la trace à l'instant courant mais elle pourrait encore être violée dans le futur par une extension de la trace.
- **En attente** (*Pending*) : Cet état est réservé pour les opérateurs forts. Il signifie qu'aucune violation de la propriété n'est survenue jusqu'à l'instant présent mais que les obligations futures n'ont pas encore été rencontrées. La propriété peut alors être satisfaite ou violée par l'extension de la trace.
- **Échouée** (*Fail*) : C'est un état définitif qui signifie qu'un état qui ne satisfait pas la propriété a été rencontré. La propriété ne pourra être satisfaite sur aucune extension du chemin d'exécution.

3.1.2 La sémantique de PSL

La sémantique des formules PSL est définie selon un modèle $M = (S, S_0, R, P, L)$ où S est l'ensemble des états finis du modèle M , $S_0 \subseteq S$ est l'ensemble des états initiaux, $R \subseteq S \times S$ est la relation de transition, P est un ensemble non vide de propositions atomiques. Les expressions booléennes sont la combinaison d'une ou plusieurs propositions atomiques. Une trace d'évaluation possible est une séquence finie ou infinie d'états $\pi = (\pi_0, \pi_1, \pi_2, \dots)$ telle que $(\pi_i, \pi_{i+1}) \in R$. Dans le cadre de la simulation, les traces sont finies. Chaque état correspond à un ensemble de propositions atomiques valides dans cet état. Cette correspondance est faite via la fonction $L : S \rightarrow 2^P$. Par conséquent, la trace est un mot de $\Sigma = 2^P \cup \{\top, \perp\}$. Pour chaque expression booléenne b , \top satisfait toujours b , et \perp est l'élément inverse tel que \perp ne satisfait jamais b .

Un mot $w \subseteq \Sigma$ est un ensemble de lettres dont chacune correspond à un cycle d'évaluation décrivant l'état du modèle à cet instant précis. Une lettre est par conséquent, une composition des propositions atomiques sur un cycle d'évaluation. Le mot $w = (\ell_0, \ell_1, \dots, \ell_n)$ correspond à une trace finie de longueur $n + 1$ dénotée $|w|$. Dans les exemples des traces d'évaluation des propriétés P_1 , P_2 et P_3 , les mots sont de taille 6.

Soient $i, j \in \mathbb{N}$ deux indices tels que $j < |w|$ et $i \leq j$. La lettre ℓ_{i-1} , notée w^{i-1} dénote le $i^{\text{ème}}$ cycle d'évaluation et $w^{i..}$ indique le suffixe de la trace w commençant par la lettre w^i . Par induction, $w^{i..j}$ est la portion de la trace w délimitée par les lettres w^i et w^j . Prenons par exemple la trace de la propriété P_1 donnée dans la Figure 3.2, la lettre ℓ_2 est composée des propositions atomiques $\{req, \neg ack\}$, ce qui signifie que ℓ_2 **satisfait** la proposition composée de la conjonction logique de deux propositions atomiques req et $\neg ack$. La relation de satisfaction est notée :

- \models dans le cas général,
- \models_B quand elle est appliquée dans le domaine B des expressions booléennes,
- \models_F pour les formules FL.

L'expression est alors notée $(\ell_2) \models_B (req \wedge \neg ack)$. On déduit par définition que $(\ell_2) \models_B req$ et $(\ell_2) \models_B \neg ack \stackrel{def}{=} (\ell_2) \models_B req$ et $(\ell_2) \not\models_B ack$. Les règles utilisées sont définies dans [PSL05, PSL10] pour le cas général.

Les lettres sont le résultat de l'échantillonnage d'une trace d'exécution dans le temps. Dans le cadre d'une simulation, l'extraction des cycles d'évaluation (lettres) est relatif au simulateur. Ainsi, la trace (ou mot) pourrait varier d'un outil à un autre selon la sémantique de simulation (continue ou basée sur les événements). Afin d'éviter ce comportement, PSL fournit un moyen pour déterminer explicitement la référence de l'échantillonnage de la trace représenté par l'opérateur "@". Pour un modèle RTL synchrone sur les fronts montants d'une horloge clk , par exemple, l'utilisateur peut indiquer que l'échantillonnage de la trace d'exécution de la propriété se fera sur les fronts montants de clk via l'expression PSL $@(\text{clock_expression})$ où clock_expression pourrait être :

- une expression booléenne de l'horloge dont la syntaxe dépend du langage HDL sous jacent. Dans le cas d'un modèle décrit en VHDL, l'expression booléenne sera `@(clk'event and clk='1')` ou `@(rising_edge(clk))`⁸.
- l'une des fonctions PSL de la couche booléenne dites “*Built-in-function*” telles que `rose(clk)` pour spécifier le front montant de `clk` ou `fell (clk)` pour le front descendant[PSL05].

3.1.2.1 La sémantique des opérateurs temporels FL

L'ensemble FL contient les opérateurs temporels suivants :**always**, **never**, **eventually!**, **before**, **until**, **next**, **next[i]**, **next.a[i..j]**, **next.e[i..j]**, **next.event**, **next.event[i]**, **next.event.a[i..j]**, **next.event.e[i..j]**. Certains de ces opérateurs existent aussi en version forte (*c.f.* paragraphe 3.1.1.2) et en version recouvrante⁹. Ces derniers ne seront pas utilisés dans ce manuscrit.

La sémantique des formules FL est définie de manière inductive sur la base de la sémantique des expressions booléennes. L'analyse de la sémantique formelle¹⁰ définie dans le standard [PSL05] montre que tout opérateur FL temporel peut être exprimé grâce aux opérateurs **next!** et **until!**. Nous proposons de détailler la sémantique des opérateurs clés **next!** et **until!** et de donner la sémantique des principaux opérateurs temporels utilisés dans cette thèse en fonction de ces deux derniers. Pour le reste des opérateurs non mentionnés, les définitions sont disponibles dans [PSL05, PSL10].

Pour les explications à suivre, nous allons considérer que φ , φ_1 , φ_2 et ψ sont des formules FL.

L'opérateur **next!** demande que son opérande soit satisfait impérativement au prochain cycle d'évaluation de la trace d'exécution. Dans la version faible, la présence du prochain cycle d'évaluation n'est pas obligatoire. La sémantique formelle des deux versions est définie comme suit (formules 3.1a et 3.1b) :

$$v \models_F \mathbf{next!} \psi \Leftrightarrow |v| > 1 \text{ et } v^{1..} \models_F \psi \quad (3.1a)$$

$$v \models_F \mathbf{next} \psi \stackrel{def}{=} \neg \mathbf{next!} \neg \psi \quad (3.1b)$$

Plusieurs variantes de l'opérateur **next** sont proposées pour donner plus de flexibilité dans les contraintes temporelles. On cite par exemple **next[i]** satisfaite si l'opérande est satisfait après i cycles, **next.a[i..j]** nécessitant que l'opérande soit satisfait dans tous les cycles d'évaluation entre le i ème et j ème prochains cycles, et **next.e[i..j]** satisfaite s'il existe un cycle d'évaluation entre le i ème et le j ème cycle dans lequel l'opérande est satisfait.

⁸recommandé depuis 1993

⁹Si l'opérande gauche est satisfait alors il doit le rester jusqu'au premier cycle d'évaluation dans lequel l'opérande droit est satisfait.

¹⁰sémantique exprimée dans un formalisme mathématique

Intuitivement, l'opérateur **next!**[i], en particulier, coïncide avec l'opérateur **next!** dans le cas où $i = 1$.

L'opérateur fort **until!** est un opérateur FL binaire. La satisfaction de la formule “**left_operand until! right_operand**” sur la trace v requiert que l'opérande gauche soit satisfait jusqu'à production impérative de l'opérande droit. Ainsi, dans la version forte, l'accent est porté sur l'opérande droit qui représente la condition de terminaison.

Reprenons l'exemple de la propriété P_1 et supposons en outre que la spécification indique que la requête doit se maintenir jusqu'à la réception de l'acquittement. La propriété correspondante est $P_4 = \mathbf{always}(\text{req} \rightarrow \text{req until! ack})$. Nous proposons trois traces d'exécution données dans la Figure 3.5 dont la première (a) satisfait la propriété contrairement aux deux autres. On note que la deuxième trace (b) pourrait modéliser P_4 avec la version faible de l'opérateur **until!**. La vérification de la trace (c) échoue à l'instant #4.

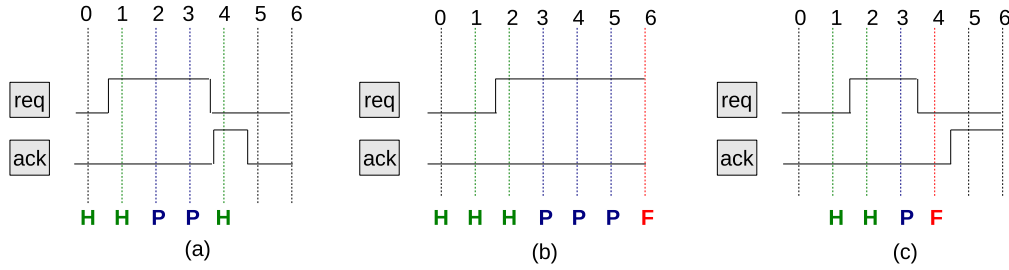


FIGURE 3.5 – Exemples de traces pour la propriété P_4

La sémantique s'exprime formellement comme suit (formules 3.2a et 3.2b) :

$$v \models_F \varphi_1 \mathbf{until!} \varphi_2 \Leftrightarrow \exists k < |v| \text{ tel que } v^{k..} \models_F \varphi_2 \text{ et } \forall j < k, v^{j..} \models_F \varphi_1 \quad (3.2a)$$

$$\varphi_1 \mathbf{until} \varphi_2 \stackrel{def}{=} (\varphi_1 \mathbf{until!} \varphi_2) \vee \mathbf{always} \varphi_1 \quad (3.2b)$$

Dans PSL_{ss} , l'opérande droit φ_2 doit être booléen. Aucune contrainte n'est imposée pour l'opérande gauche.

Définitions formelles basée sur **until!** :

Comme on l'a déjà vu dans les exemples précédents, l'opérateur **always** est un opérateur unaire qui déclenche l'évaluation de la formule φ à chaque instant de la trace v . Dans LTL, on dit que la formule doit être vraie globalement sur toute la trace. La définition formelle est donnée en fonction de celle de l'opérateur **until!** comme suit (formule 3.3) :

$$\mathbf{always}(\varphi) \stackrel{def}{=} \neg(\text{true until! } \neg\varphi) \quad (3.3)$$

L'opérateur **eventually!** est aussi un opérateur unaire qui spécifie que l'opérande doit avoir lieu dans le futur. Si on reprend notre propriété de vivacité P_1 en considérant que

chaque requête doit être suivie dans le futur par un acquittement alors on pourra énoncer la propriété $P_5 = \text{always } (\text{req} \rightarrow \text{eventually! ack})$. Dans la spécification de la propriété, rien n'empêche plusieurs requêtes d'être validées par le même acquittement. Ainsi, les traces illustrées par les Figures 3.5(a) et 3.5(c) satisfont la propriété P_5 . Dans la trace illustrée par la Figure 3.5(b) La propriété P_4 est dans l'état Pending dès la satisfaction de l'opérande gauche de l'opérateur **until!** et jusqu'à la fin de cette trace finie dans laquelle *ack* ne se produit pas.

Les propriétés P_4 et P_5 décrivent des comportements différents : P_5 ne requiert pas que la requête soit maintenue. La trace donnée par la Figure 3.6 en témoigne. Elle satisfait P_5 mais pas P_4 .

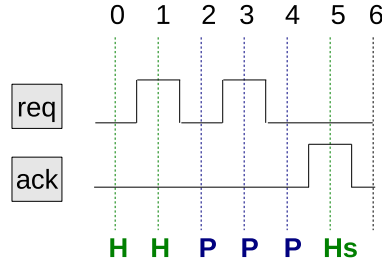


FIGURE 3.6 – Exemples de trace qui satisfont la propriété P_5

Ces propos sont aussi confirmés par la définition formelle de la sémantique donnée dans la formule 3.4 :

$$\text{eventually! } \varphi \stackrel{\text{def}}{=} \text{true until! } \varphi \quad (3.4)$$

Dans PSL_{ss} , l'opérande doit être une expression booléenne ou une SERE.

L'opérateur **before!** est un opérateur binaire. Il requiert que l'opérande gauche ait lieu avant l'opérande droit. Contrairement à l'opérateur **until!**, l'accent est porté sur l'opérande gauche. Par exemple, La propriété P_6 spécifie que “si on a une requête *req* à l'instant courant alors la satisfaction de *ack* est nécessaire avant la prochaine requête”. Les deux versions de la propriétés sont formulées dans 3.5 et 3.6. L'utilisation de l'opérateur **next** permet d'exclure l'instant courant durant lequel l'expression *req* est vrai dans l'évaluation de l'opérateur **before**. Dans la version faible (P_6-w), la propriété est satisfaite par les deux traces proposées dans la Figure 3.7. Avec la version forte (P_6-s), la propriété n'est pas satisfaite par la trace (a) car *ack* n'apparaît pas après la deuxième occurrence de *req*.

$$P_6-s : \text{always} (\text{req} \rightarrow \text{next} (\text{ack} \text{ before! req})) \quad (3.5)$$

$$P_6-w : \text{always} (\text{req} \rightarrow \text{next} (\text{ack} \text{ before req})) \quad (3.6)$$

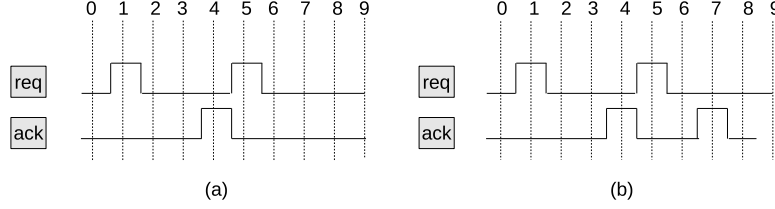


FIGURE 3.7 – Exemples de trace qui satisfait la propriété P_6

La définition formelle de la version forte et faible de cet opérateur est donnée par les équations 3.7a et 3.7b comme suit :

$$\varphi \text{ before! } \psi \stackrel{def}{=} \neg \psi \text{ until! } (\varphi \wedge \neg \psi) \quad (3.7a)$$

$$\varphi \text{ before } \psi \stackrel{def}{=} \neg \psi \text{ until } (\varphi \wedge \neg \psi) \quad (3.7b)$$

Dans PSL_{ss} , les deux opérandes doivent être booléens.

L'opérateur **next_event!**(b)(φ) indique que la formule φ doit être satisfaite par la trace à la prochaine satisfaction de l'expression booléenne b . Nous donnons les définitions formelles des versions forte et faible de cet opérateur respectivement dans les formules 3.8a et 3.8b :

$$\text{next_event!}(b)(\varphi) \stackrel{def}{=} \neg b \text{ until! } (b \wedge \varphi) \quad (3.8a)$$

$$\text{next_event}(b)(\varphi) \stackrel{def}{=} \neg b \text{ until } (b \wedge \varphi) \quad (3.8b)$$

avec $v \models_F b \wedge \varphi \Leftrightarrow v \models_B b$ et $v \models_F \varphi$

3.1.2.2 La sémantique des SERES

L'évaluation des expressions séquentielles régulières se fait sur une trace finie. Pour la suite des explications, nous considérons en plus :

- $v = \epsilon$ est la trace vide, de taille $|v| = 0$,
- \bar{v} est le dual de v obtenu en inversant tous les \top et \perp ,
- r, r_1, r_2 trois SERES synchrones.

Nous allons considérer les définitions formelles de la sémantique d'exécution des opérateurs SERE qui vont être utilisés dans les chapitres suivants à savoir : la séquentialité,

la répétition consécutive et l'implication suffixe.

Les opérateurs de séquence

L'opérateur SERE de concaténation “;” est un opérateur reliant deux SEREs r_1 et r_2 . La séquence “ $\{r_1; r_2\}$ ” est satisfaite sur la trace finie v s'il existe une décomposition de la trace v en deux sous traces v_1 et v_2 telle que r_1 est satisfaite sur v_1 et r_2 est satisfaite sur v_2 . La sémantique formelle de trace donnée dans [PSL10] est donnée par l'équation 3.9.

$$v \models r_1; r_2 \Leftrightarrow \exists v_1, v_2 \text{ s.t. } v = v_1 v_2, v_1 \models r_1, \text{ and } v_2 \models r_2 \quad (3.9)$$

L'opérateur SERE de fusion “:” est aussi un opérateur binaire. Cet opérateur représente une concaténation recouvrante sur un cycle de la trace. La définition de la sémantique formelle de trace de cet opérateur selon la norme PSL est donnée par l'équation 3.10 :

$$v \models r_1 : r_2 \Leftrightarrow \exists v_1, v_2 \text{ and } l, \text{ s.t. } v = v_1 l v_2, v_1 l \models r_1, \text{ and } l v_2 \models r_2 \quad (3.10)$$

Les opérateurs de répétition

Classiquement pour les expressions régulières, $r[+]$ représente la répétition de l'expression r 1 ou plusieurs fois et $r[*]$ représente la répétition de r 0 ou plusieurs fois. PSL offre des extensions comme $r[*i]$ qui signifie que r doit être se répéter i fois et $r[*i..j]$ indiquant que r doit se répéter entre i et j fois. Les définitions formelles sont données par les équations suivantes (c.f. 3.11) :

$$\begin{aligned} v \models [*0] &\Leftrightarrow v = \epsilon \\ v \models r[*] &\Leftrightarrow \text{either } v \models [*0] \text{ or } \exists v_1, v_2 \text{ s.t. } v_1 \neq \epsilon, v = v_1 v_2, v_1 \models r \text{ and } v_2 \models r[*] \\ [*] &\stackrel{\text{def}}{=} \text{true}[*] \\ v \models r[+] &\Leftrightarrow v \models r; r[*] \\ r[*i] &\stackrel{\text{def}}{=} \overbrace{r r \dots r}^i \\ r[*i..j] &\stackrel{\text{def}}{=} r[*i] \mid \dots \mid r[*j] \\ r[*i..inf] &\stackrel{\text{def}}{=} r[*i]; r[*] \end{aligned} \quad (3.11)$$

L'implication suffixe

Cet opérateur s distingue de l'implication logique par le fait que son opérande gauche est une SERE. Il existe deux variantes : l'implication suffixe recouvrant “|->” et non

recouvrant “ \models ”.

La sémantique informelle de l'implication suffixe recouvrant est identique à celle de l'opérateur d'implication logique. La satisfaction de l'implication suffixe non recouvrant nécessite en revanche que la formule FL φ soit satisfaite un cycle après la satisfaction de la séquence r . Les équations 3.12a et 3.12b donnent les définitions de la sémantique formelle de ces deux opérateurs :

$$v \models r \mid\!\!\rightarrow \varphi \Leftrightarrow \forall j < |v| \text{ s.t. } \bar{v}^{0..j} \models r, v^j \models_F \varphi \quad (3.12a)$$

$$r \models\!\!\rightarrow \varphi \stackrel{def}{=} \{r; true\} \mid\!\!\rightarrow \varphi \quad (3.12b)$$

Nous considérons l'exemple de l'assertion :

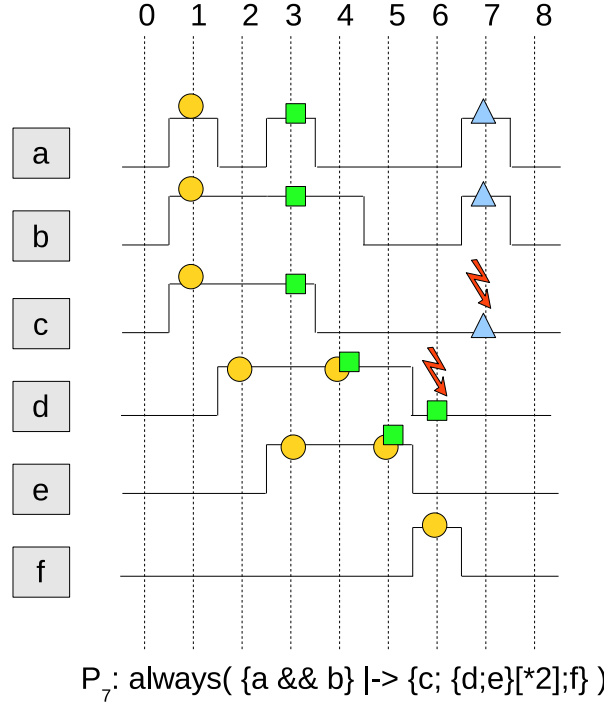
$P_7 = \text{always}(\{a \ \&\& \ b\} \mid\!\!\rightarrow \{c; \{d; e\}[*2]; f\})$.

Cette assertion spécifie le comportement suivant : “*si a et b sont satisfaits à l'instant courant alors c doit être satisfaite au même instant suivie à l'instant suivant de deux occurrences consécutives de la séquence composée de d suivie à l'instant suivant de e . À l'instant suivant la production de ces deux occurrences, le motif f doit se produire*”. L'opérateur **always** permet de redémarrer l'évaluation de la propriété à chaque instant de la trace. Ainsi, chaque occurrence de l'expression $\{a \ \&\& \ b\}$ permettra le démarrage d'une instance d'évaluation indépendante. Dans l'exemple de trace illustrée par la Figure 3.8, chaque instance d'évaluation est marquée par une forme particulière (cercle, carré, triangle). Cette figure décrit l'évolution de trois instances d'évaluation. La première instance dénotée par un cercle démarre à l'instant #1 et elle est satisfaite à l'instant #6. La deuxième dénotée par un carré démarre à l'instant #3 et elle est violée à l'instant #6 car le motif d ne s'est pas produit au cycle suivant la production du motif e . La troisième instance dénotée par un triangle est violée à cause de l'absence du motif c au cycle d'évaluation #7.

3.2 La vérification dans les flots de conception

Dans le cadre d'une démarche de vérification basée sur les assertions, les spécifications sont décrites formellement par un ensemble de propriétés. L'ensemble des assertions forme alors une spécification formelle exécutable qui constitue la référence dans le processus de vérification. La validation du modèle de référence se fait alors par rapport à cette spécification formelle.

Nous pouvons remarquer à travers l'aperçu donné du langage PSL que la sémantique de trace se prête plus facilement à des descriptions de modèles synchrones au niveau RTL. En effet, PSL offre naturellement les fonctions nécessaires pour permettre l'échantillon-

FIGURE 3.8 – Exemple de trace pour la propriété P_7

nage des traces sur les fronts des horloges. De plus, la couche booléenne est particulièrement adaptée au support des signaux.

3.2.1 La vérification au niveau RTL : Horus

Comme on l'a vu dans le premier chapitre (*c.f.* section 1.2.4.1), plusieurs solutions ont été proposées pour l'implémentation d'une méthodologie de vérification à base d'assertions au niveau RTL. Dans cette section, on s'intéresse en particulier à l'outil Horus qui a été développé au laboratoire TIMA et qui constitue la solution de vérification au niveau RTL de notre flot de conception. On évoquera avec plus de précision, la méthode de construction des moniteurs de surveillance à partir des assertions PSL.

En s'appuyant sur les travaux présentés dans la section 1.2.4.1 portant sur les techniques de génération des moniteurs de surveillance au niveau RTL à partir des assertions, on peut conclure que l'approche classique consiste globalement à traduire les propriétés en automates finis plus ou moins optimisés et à générer les moniteurs à partir de ces automates. La solution adoptée dans Horus est différente[OMAB08]. Elle s'appuie sur une approche qui permet la construction du moniteur de la propriété PSL à partir de son arbre syntaxique : à chaque opérateur FL de PSL correspond un moniteur élémentaire (ou primitif) paramétrable qui obéit aux contraintes de PSL_{ss} . La taille du moniteur final est alors linéaire par rapport au nombre d'opérateurs de la propriété. Il s'agit d'une méthode de construction modulaire, les blocs de base sont les moniteurs élémentaires codés en

VHDL qui forment une bibliothèque de composants réutilisables. L'interconnexion des instances de ces modules forme le moniteur global.

Afin de faciliter l'interconnexion entre eux, les moniteurs élémentaires se conforment à une même structure[OMAB08].

Au niveau de l'interface, le moniteur possède jusqu'à cinq ports d'entrée et trois ports de sortie. Mis à part le signal d'horloge *clk* et de remise à zéro (*reset*), les trois autres entrées sont :

- *start* : c'est un signal d'activation en provenance du composant qui précède le moniteur. L'activation du moniteur permet le démarrage de l'évaluation de l'opérateur correspondant.
- *cond* et *expr* : ces entrées sont les opérandes de l'opérateur implémenté par le moniteur élémentaire. Si l'opérateur implémenté est unaire, alors le moniteur correspondant aura une seule entrée *expr*.

Les sorties sont :

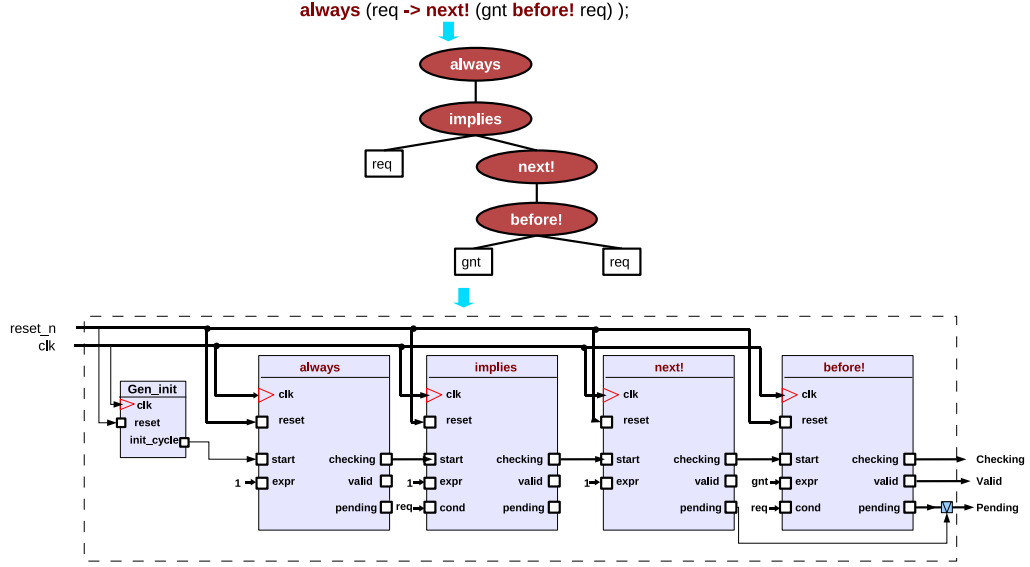
- *Checking* : quand il est à '1', ce signal indique que le moniteur est actif et que la sortie *Valid* sera effective au prochain front d'horloge. Cette sortie est liée à l'entrée *start* du moniteur suivant, s'il existe.
- *Valid* : ce signal donne le résultat d'évaluation en fonction des valeurs des opérandes en entrée. Quand le moniteur est actif, la valeur '0' indique une erreur et '1' signifie le respect de l'opérateur implémenté.
- *Pending* : Cette sortie indique que le moniteur a été activé mais que la condition de satisfaction n'est pas encore atteinte. Cette sortie est considérée uniquement pour les versions fortes des opérateurs PSL.

Les sorties *Valid* et *Checking* du moniteur global correspondent aux sorties du dernier moniteur élémentaire correspondant à l'opérateur FL le plus à droite de la formule. La sortie *Pending*, par contre, est la disjonction des sorties *Pending* de tous les opérateurs forts de la formule.

À titre illustratif, on propose de reprendre l'exemple de la propriété P_6 en remplaçant l'opérateur **next** par sa version forte. La propriété devient $P_8 = \text{always}(\text{req} \rightarrow \text{next!}(\text{gnt before! req}))$. Nous proposons de construire le moniteur généré par Horus. La correspondance entre l'arbre syntaxique et le moniteur global est donnée par la Figure 3.9.

Le code du moniteur VHDL généré est indépendant du modèle à vérifier (DUV¹¹). La liaison entre les signaux du modèle à vérifier et les entrées du moniteur est faite au moyen d'un "port map" VHDL. L'horloge et le reset du moniteur sont connectés à l'horloge et au reset du modèle à vérifier. Ainsi, l'échantillonnage de la trace des moniteurs est conforme à l'échantillonnage de la trace d'exécution du modèle à vérifier. Le résultat obtenu est le

¹¹Design Under Verification

FIGURE 3.9 – Moniteur RTL généré par Horus pour la propriété P_8

code VHDL synthétisable contenant à la fois le modèle à vérifier et tous les moniteurs connectés. Ce code peut être simulé ou synthétisé. La construction des moniteurs dans l'outil ISIS présenté à la section 3.2.2.3 sera basée sur un principe similaire.

3.2.2 La vérification au niveau TLM

La validation du modèle TLM est primordiale. C'est le modèle de référence pour le reste du flot de conception descendant. L'utilisation de l'ABV dynamique pour la vérification des modèles au niveau système est une technique relativement récente. Peu d'outils industriels et de travaux de recherche ont été réalisés. La principale raison, est que les langages de spécification des propriétés temporelles (PSL et SVA) sont plus adaptés pour le niveau RTL. En effet, ces langages ont été principalement conçus pour la spécification formelle au niveau RTL.

3.2.2.1 Les approches au niveau *Cycle Accurate*

Plusieurs travaux ont été proposés dans le but de prouver la possibilité d'appliquer les techniques de l'ABV sur des descriptions SystemC. Certains sont simplement le résultat d'une migration des solutions de l'ABV du niveau RTL de VHDL ou Verilog vers des descriptions SystemC au niveau *cycle-accurate*. Parmi ces travaux, on cite une extension de l'outil FoCs[ABG⁺00] pour la génération de moniteurs C++ selon la même démarche que celle suivie pour la génération des moniteurs VHDL et Verilog. Ces moniteurs ne sont pas adaptés pour le niveau transactionnel.

En 2001, **Ruf, Rosenstiel et al.** proposaient dans [RHKR01] et [BGR02] une ap-

proche de l'ABV appliquée à des descriptions SystemC au niveau signal. Bien avant la standardisation de PSL, les propriétés ont été décrites en utilisant une version du langage LTL réduite sur les traces finies dite FLTL¹². Ces propriétés sont ensuite traduites en automates finis non déterministes. Les processus de déterminisation et de réduction permettent d'obtenir un automate particulier appelé AR-automaton à deux états finaux : **A** pour “*Acceptance*” et **R** pour “*Reject*” dans le but de signaler la satisfaction et la violation de la propriété. Les moniteurs générés à partir de ces automates sont des processus SystemC standards qui viennent surcharger le moteur de simulation SystemC. De plus, l'instrumentation du modèle à surveiller se fait par l'utilisateur. Ce dernier a la responsabilité d'identifier l'endroit approprié pour introduire les assertions FLTL au moyen d'une surcharge de la macro `sc_assert()` de SystemC. La position de la formule est importante car elle doit se situer après l'événement susceptible de déclencher l'évaluation. Cette approche suppose alors un ordre total des événements. Or cet ordre est difficile à déterminer à l'avance dans le cadre d'une simulation non déterministe telle qu'elle peut être produite par le moteur de simulation SystemC [SC005]. Ainsi, il est possible d'avoir des fausses violations dûes à un quelconque changement du code du modèle. De plus, le processus de pré-traitement qui précède la génération des moniteurs nécessite la consommation de beaucoup de ressources mémoire. Cependant, une fois produits, les automates sont sauvegardés dans une base de données pour être réutilisés ultérieurement.

Habibi et Tahar proposent dans [HT05][HT06] une méthode de vérification intégrée dans les activités de conception. Les propriétés PSL ainsi que le modèle SystemC sont modélisés par des diagrammes UML[OMG09]. D'un côté le modèle est décrit par des diagrammes de cas d'utilisation, de séquence et de classe. De l'autre côté, les propriétés sont représentées par des diagrammes de séquence UML modifiés afin de pouvoir inclure des annotations temporelles (horloge, durée, retard, etc.), des opérateurs temporels (always, next, eventually, until) et des opérateurs dits de séquence (next, prev, etc.). Les auteurs ne détaillent pas la sémantique de ces opérateurs.

UML est un langage de spécification graphique. L'objectif de la modélisation des propriétés par des diagrammes de séquence est de faciliter leur spécification au moyen d'une représentation graphique des séquences de messages qui constituent ces propriétés. L'inconvénient majeur est que la conservation de la sémantique formelle de PSL n'est pas garantie en utilisant un langage de description informel.

Les représentations graphiques sont ensuite transformées en machines à états abstraites ASM¹³, décrite en AsmL¹⁴. Ces spécifications formelles sont utilisées pour la génération

¹²Finite Linear time Temporal Logic

¹³Abstract State Machine

¹⁴*Abstract State Machine Language*, un langage de spécification formelle exécutable de Microsoft (<http://research.microsoft.com/en-us/projects/asml>)

des FSMs des propriétés. Durant cette étape, l'ensemble des propriétés est vérifié statiquement par *Model Checking*. En cas d'erreur, le flot de conception du système doit être repris dès la première étape de modélisation UML. Cette approche est caractérisée par les limitations des techniques de vérification statique à base de FSMs. Pour les systèmes complexes, il est difficilement envisageable de réaliser une vérification étendue.

Le modèle SystemC est ensuite généré à partir de code AsmL d'un côté, et les propriétés sont traduites en modules codés en Microsoft Visual C#¹⁵ de l'autre côté. Une vérification dynamique peut alors être réalisée en simulant conjointement les moniteurs C# et le modèle SystemC. Les détails concernant la génération des moniteurs et leur intégration dans le modèle SystemC ne sont pas donnés. De plus, les exemples de propriétés traitées portent tous sur des descriptions SystemC *cycle-accurate*, au niveau signal.

Dans [NH06], **Niemann et al.** proposent une méthodologie pour l'utilisation de l'ABV dynamique pour des modèles TLM. Leur approche est basée sur la description de propriétés SVA transactionnelles sous la forme d'une implication logique entre une condition booléenne de déclenchement et l'expression booléenne à évaluer. Toute transaction sera remplacée par un signal booléen positionné à '1' si elle est active et à '0' sinon. En raison des délais supplémentaires nécessaires pour la mise à jours des signaux, il n'est pas possible de prendre en considération les transferts bloquants.

En premier lieu, les auteurs ont recours à des techniques de la programmation orientée aspects¹⁶ pour constituer la trace de toutes les transactions sous forme des signaux booléens correspondants. Ensuite, le modèle SystemC est simulé pour générer une trace VCD¹⁷ des signaux booléens. Les assertions SVA sont appliquées au modèle Verilog produit à partir du fichier VCD. Cette solution est limitée car elle ne considère qu'une vision restreinte des transactions : seules les relations entre les transactions sont prises en considération mais toutes les données de la transaction sont ignorées. Une horloge globale artificielle est incluse afin de permettre l'échantillonnage des traces.

3.2.2.2 Les approches au niveau transactionnel

Dans sa thèse [Lah06], **Lahbib** a choisi le langage PSL pour la description des propriétés au niveau transactionnel. En effet, PSL serait le langage le mieux adapté pour la vérification des modèles SystemC-TLM en raison de sa sémantique basée sur des traces pouvant être associées à tout événement. Cependant, une adaptation du langage est nécessaire dans ce travail pour supporter l'abstraction des données de communications.

¹⁵Langage orienté objet développé sur la base de C++ par Microsoft au sein de l'initiative .Net et approuvé comme un standard ISO en 2006 (ISO/IEC 23270 :2006) <http://msdn.microsoft.com/en-us/vstudio/hh341490.aspx>

¹⁶<http://www-igm.univ-mlv.fr/~dr/XPOSE2005/gmasquelier/contents/presentation.html>

¹⁷Value Change Dump

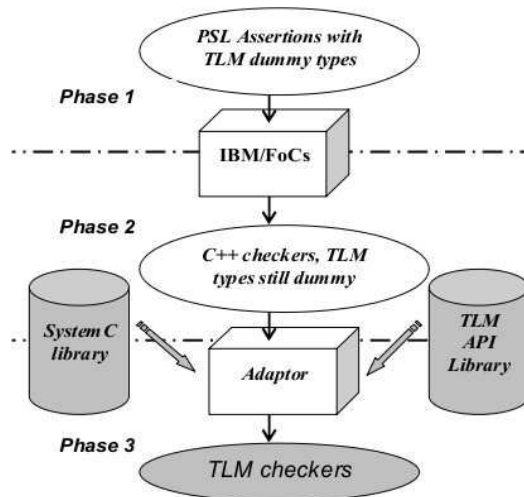


FIGURE 3.10 – Le flot de la génération des moniteurs proposé dans [Lah06]

Le flot de génération des moniteurs transactionnels décrit par la Figure 3.10 s’appuie sur les moniteurs FoCs pour la génération du code C++ à partir des propriétés PSL. Afin de permettre la spécification des propriétés transactionnelles avec PSL, les transactions et les données de communication sont annotées par des variables booléennes fictives déclarées dans la couche modélisation des assertions. Après la génération des moniteurs C++, ces variables sont remplacées par les types TLM effectifs par un composant dédié dit “Adaptateur”. Les moniteurs sont ensuite inclus dans des enveloppes (*wrapper*) qui permettent de contrôler le démarrage des évaluations au cours de la simulation. La solution proposée possède deux atouts majeurs : le mécanisme d’observation et l’échantillonnage de la trace de simulation qui est fait non pas par les cycles de l’horloge mais par les événements liés aux transactions. Cependant, elle présente aussi quelques limitations :

- Les fonctions et les données supportées appartiennent uniquement à la couche transport de TLM.
- La solution n’est applicable que sur les descriptions TLM les plus abstraites. Les descriptions hybrides où des modules au niveau signal peuvent être inclus ne sont pas prises en considération.
- Les propriétés portant sur plusieurs canaux de communication ne sont pas supportées.
- Les temps de simulation peuvent être sévèrement affectés par l’activation des moniteurs. En effet, les moniteurs C++ générés par FoCs ne sont pas optimisés et leur taille peut augmenter de manière importante en fonction de la complexité des formules PSL.

Dans [EEH⁺06][Ese08], **Ecker, Esen et al.** proposent un nouveau langage de spécification des propriétés transactionnelles. Ils estiment que les langages existants (SVA, PSL) ne sont pas adaptés à la spécification des propriétés transactionnelles. La sémantique de

ces langages ne permettrait pas l'identification explicite des événements transactionnels et des relations de corrélation entre ces événements. Ils considèrent que chaque transaction est caractérisée par un événement de début et de fin et par une durée d'exécution.

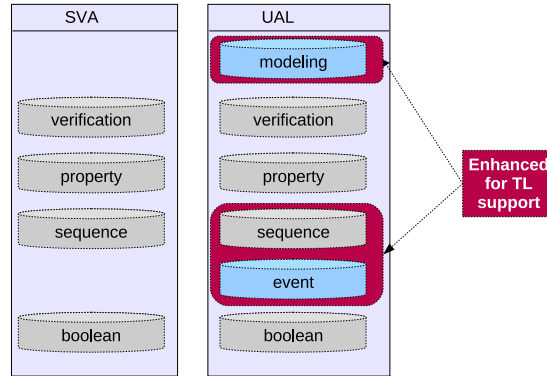


FIGURE 3.11 – L'architecture du langage UAL

Afin d'exprimer les événements transactionnels, les auteurs ont choisi d'étendre la sémantique du langage SVA[SV009] en ajoutant deux couches supplémentaires comme l'illustre la Figure 3.11. Le langage résultant est dit UAL¹⁸ et inclut :

- La couche modélisation : cette couche est inspirée de la couche de modélisation de PSL. Elle est utilisée pour introduire des variables auxiliaires dans les assertions UAL.
- La couche événement : cette couche définit un ensemble d'opérateurs permettant de générer des événements nommés *trigger* qui activent l'évaluation des séquences. Dans UAL, l'ensemble E des événements est composé de tous les événements SystemC initialement présents dans le modèle à vérifier auxquels s'ajoutent les événements de début et de fin de chaque transaction générés par les opérateurs UAL. Parmi les opérateurs définis dans cette couche, nous citons par exemple les opérateurs 'START et 'END permettant respectivement de créer les événements de début et de fin des transactions. Les auteurs définissent un opérateur particulier nommé opérateur de contrainte et noté $EV_EXPR@ (BE)$. Comme son nom l'indique, cet opérateur permet de spécifier une condition booléenne (BE) pour le déclenchement de l'événement associé à l'expression d'événement EV_EXPR . Le démarrage de l'évaluation nécessite alors la production de l'expression d'événement et la satisfaction de la condition booléenne.
- Une extension de la sémantique de la couche séquence de SVA a aussi été proposée, afin de l'adapter aux spécificités des modèles transactionnels. Le déclenchement de l'évaluation des séquences se fait en se référant à l'opérateur de délai ($\#$). C'est un opérateur clé du langage car sa sémantique est définie indépendamment des couches

¹⁸ Universal Assertion Language

d'abstraction. Une séquence est composée par un enchainement d'opérateurs de délai qui spécifient l'ordre d'évaluation des expression booléennes.

Le langage UAL fournit une bibliothèque de moniteurs de base. Chaque moniteur est un module SystemC qui implémente un opérateur sous la forme d'un réseau de Pétri coloré¹⁹. Le moniteur global est construit automatiquement à partir de cette bibliothèque. La Figure 3.12 donne un exemple de moniteur composé de quatre séquences et d'un opérateur d'implication. Les blocs représentés par des traits continus sont les moniteurs de la bibliothèque. Ceux qui sont entourés par des traits discontinus correspondent au code automatiquement généré lié au moniteurs.

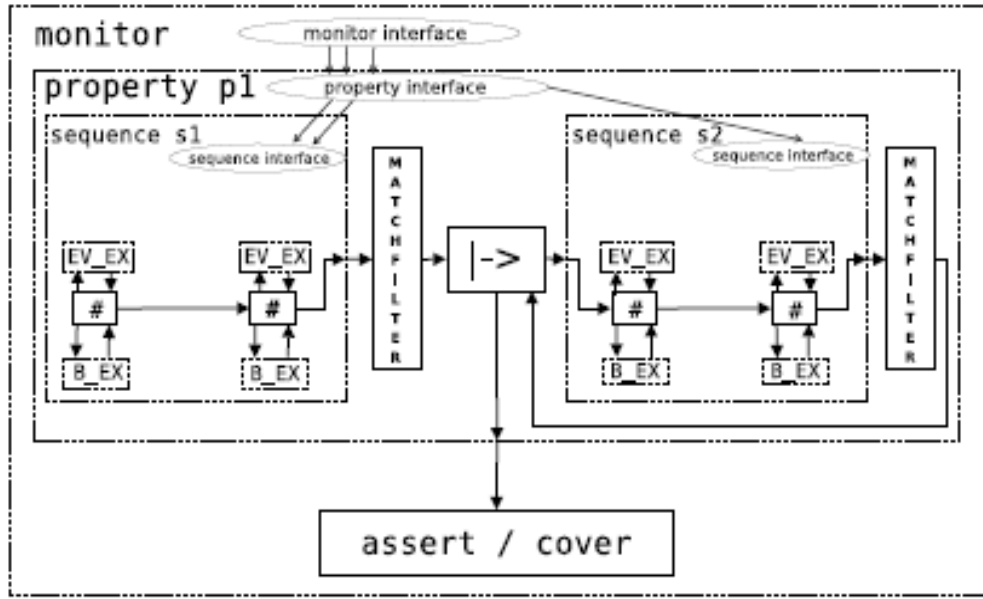


FIGURE 3.12 – Exemple d'implémentation d'un moniteur UAL [Ese08]

Chaque instance d'évaluation de la séquence est modélisée par un jeton qui se propage à travers les réseaux de pétri interconnectés. Sur la Figure, le trait continu orienté représente le sens de propagation du jeton. À chaque fois qu'un jeton atteint une expression de la couche booléenne (représentée par les boîtes B_Ex dans la Figure), l'expression est alors évaluée.

Les auteurs donnent une définition générique des opérateurs de la bibliothèque de base. Ainsi, après la génération du moniteur, l'utilisateur doit configurer son interface. De plus, l'utilisateur se charge aussi de définir le code UAL permettant d'établir les correspondances entre le moniteur et le modèle.

Afin de détecter les relations entre les transactions, il est nécessaire d'identifier le début et la fin de chaque transaction. De même que dans [Lah06], Ecker et al. expriment la nécessité d'inclure un mécanisme de temporisation et de notification. Les auteurs optent

¹⁹chaque couleur correspond à une nouvelle évaluation

pour l'introduction de modules particuliers, dits "*proxy*" qui agissent de la même façon que les enregistreurs de transactions définis dans SCV[SCV06]. Ces modules sont définis par l'utilisateur et introduits entre les composants du modèle transactionnel. Leur rôle est d'intercepter les fonctions de communication et de générer les événements associés au début et à la fin de chaque transaction. Ce mécanisme est une solution alternative à l'utilisation du modèle d'observation implémenté dans ISIS(*c.f.* page77). Pour les événements SystemC du modèle à surveiller, un processus SystemC sensible à chaque événement est créé. Un gestionnaire d'événement est chargé de recevoir les notifications de ces deux groupes de composants afin de gérer l'activation des séquences du moniteur. Un avantage de cette approche est que chaque séquence peut être sensible à un ensemble d'événements différents. Ainsi, chaque implémentation d'une séquence contient un ensemble d'observateurs d'événements. Chaque observateur est spécifique à un événement particulier de la liste de sensibilité de la séquence. Ce mécanisme autorise un degré de précision maximal lors de l'évaluation de la séquence.

En conclusion, bien qu'il ne soit pas standard, le langage de spécification des assertions défini par les auteurs est élaboré. Nous rappelons en particulier, la définition du nouvel opérateur **#**. De plus, l'infrastructure de vérification est conceptuellement très complète mais aussi complexe. Un apport non négligeable est la spécification d'une liste de sensibilité par séquence.

L'inconvénient majeur de cette approche est relatif au mécanisme d'instrumentation à base de *proxy*. Ces modules interviennent directement dans la composition du modèle et peuvent influencer le fonctionnement du système. Bien qu'ils soient à la base de toute l'infrastructure de gestion des événements, leur définition repose entièrement sur les utilisateurs.

L'automatisation n'est essentiellement présente que dans la construction des moniteurs. Par ailleurs, les différentes publications ne citent aucun exemple de propriété portant sur les communications entre plusieurs composants d'une plateforme. Tous les cas d'étude reportés montrent des assertions spécifiques à un composant en particulier. Enfin, selon les résultats cités dans [Ese08], le coût de l'instrumentation calculé en terme de temps d'exécution peut engendrer un facteur de pénalité allant de 2 à 13.

En 2010, **Große et al.** ont proposé dans [GLD10] une méthodologie de vérification statique pour les systèmes transactionnels décrits en SystemC. Il s'agit d'une amélioration de la méthode proposée dans [GD03] permettant la vérification des effets des transactions et des dépendances entre les événements des transactions. L'outil proposé supporte un sous ensemble de PSL contenant les opérateurs temporels **next** et **always** ainsi que l'implication logique. Des primitives supplémentaires sont proposées pour prendre en considération les transactions et les événements. La méthodologie implémentée est décrite

par la Figure 3.13.

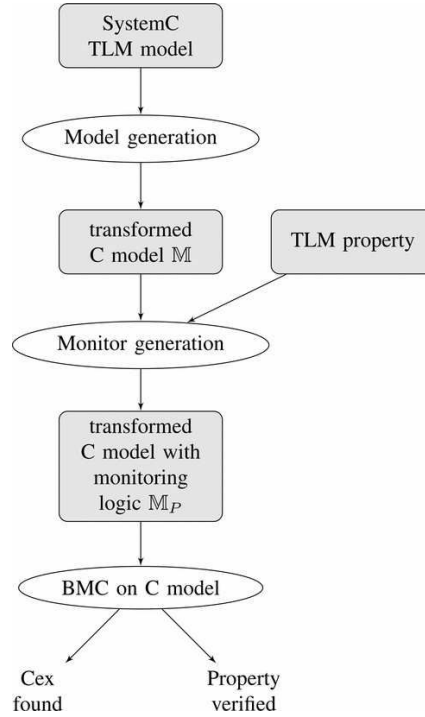


FIGURE 3.13 – Le flot de la méthodologie de vérification proposée dans [GLD10]

Le modèle SystemC est préalablement transformé en un modèle C intermédiaire dénoté \mathbb{M} . Les assertions temporelles sont modélisées en FSMs et intégrées dans le modèle intermédiaire après la génération du code C correspondant. Le modèle instrumenté est noté \mathbb{M}_P . La vérification de chaque formule est traduite en un problème décisionnel (SAT) dans lequel les valeurs des variables globales forment les différents états. Les transitions correspondent aux notifications de tous les événements du modèle. La vérification est réalisée en utilisant l'outil de *Bounded Model Checking* (CBMC)[CKL04]. Cet outil vise à trouver un chemin d'exécution de taille finie k , constituant un contre exemple de la propriété surveillée. Dans le cas contraire, la vérification continue par induction sur les chemins de taille $k + 1$ à la recherche d'un état qui ne satisfait pas la propriété. Si aucun état n'est trouvé, l'outil conclut que la propriété est satisfaite.

Bien que l'ajout des règles d'induction offre une amélioration par rapport aux solutions de vérification statiques, les résultats obtenus montrent des temps de calcul très élevés selon la complexité des propriétés surveillées. De plus, la méthode devient rapidement inefficace pour les modèles relativement complexes.

Même si les méthodes proposées dans [Lah06], [EEH⁺06], [Ese08] et [GLD10] ont des limitations, elles offrent néanmoins des réflexions intéressantes concernant notamment l'adaptation de l'utilisation de PSL pour des modèles transactionnels et les modèles d'observation des communications transactionnelles.

3.2.2.3 L'outil ISIS

L'outil ISIS[FP09][PF10] est l'implémentation d'une approche de vérification basée sur les assertions PSL pour des systèmes transactionnels développée à TIMA.

Le flot d'instrumentation du modèle SystemC

ISIS est un prototype qui permet la génération automatique des moniteurs SystemC à partir de spécifications PSL et l'instrumentation automatique du modèle à vérifier par ces moniteurs de surveillance. Comme le montre le flot d'ABV de la Figure 3.14, l'outil prend en entrée un ensemble de propriétés PSL et un modèle SystemC. L'utilisateur doit aussi indiquer les identificateurs des composants mis en jeu dans l'assertion. À partir des assertions PSL, l'outil génère automatiquement les moniteurs de surveillance associés. En se basant sur la liaison entre les composants du modèle SystemC et les identifiants, ISIS génère le code du modèle instrumenté par les instances des moniteurs générés.

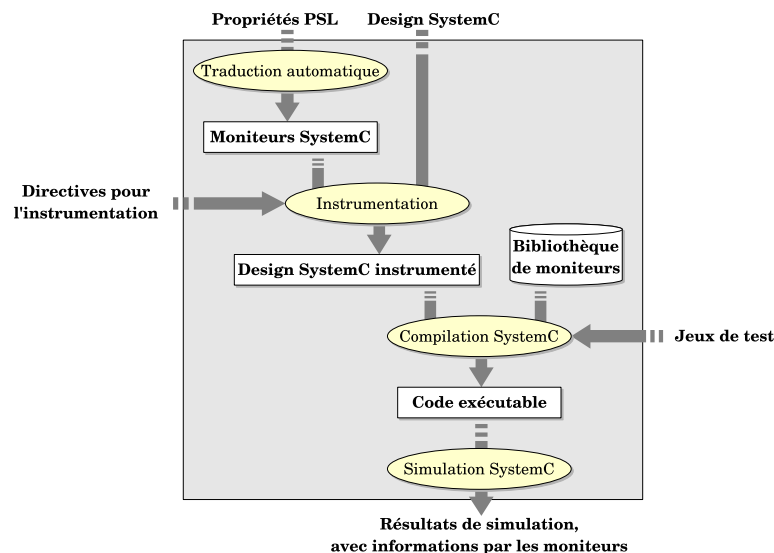


FIGURE 3.14 – Le flot pour la méthodologie de vérification proposée dans [Fer11]

Le code instrumenté du modèle est compilé en utilisant la bibliothèque de moniteurs élémentaires et la simulation SystemC du modèle à vérifier peut démarrer. Durant la simulation, les moniteurs peuvent fournir des informations pertinentes concernant les instants d'activation et les résultats d'évaluation. C'est une approche non intrusive : les propriétés PSL ne sont pas intégrées dans le modèle à vérifier et les moniteurs générés sont indépendants de ce dernier.

La génération des moniteurs ISIS

Pour construire les moniteurs SystemC dédiés à la surveillance des modèles TLM, une variante de la méthode d'interconnexion des moniteurs Horus a été conçue. Une bibliothèque de moniteurs élémentaires C++ a été initialement développée. À la différence

des moniteurs VHDL d' Horus, les moniteurs ISIS ne sont pas des composants matériels synchrones mais des objets C++ non temporisés. Chaque classe correspond à un opérateur PSL et contient un constructeur et une fonction `update()`. L'interconnexion des moniteurs ISIS élémentaires se fait alors par l'instanciation de ces classes. L'évaluation des opérateurs à chaque activation se fait au moyen de la fonction `update`.

À la place des ports d'entrée sorties des moniteurs Horus, les interfaces des moniteurs SystemC sont composées par un ensemble de variables protégées, certaines étant utilisées pour contenir les valeurs des opérandes (*expr*, *cond*) ou la valeur d'activation du moniteur (*start*) et d'autres contiennent le résultat de l'évaluation de l'opérateur courant (*valid*, *checking*, *pending*). La signification des ces variables est identique à celle des ports des moniteurs Horus décrite page 66. Les moniteurs ISIS n'ont pas de signal d'horloge en entrée, une variable booléenne “**reset**” permet la remise à zéro des moniteurs élémentaires.

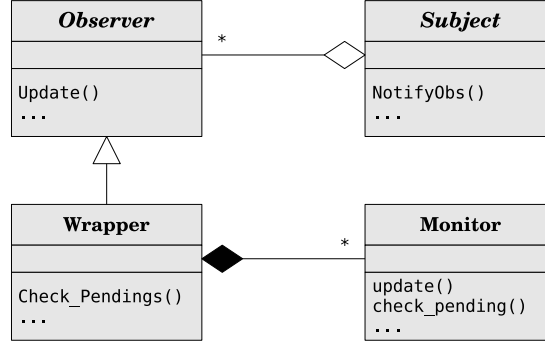
Il faut noter aussi qu'ISIS n'inclut pas de support pour les opérateurs SEREs de PSL. En effet, ces opérateurs permettant de décrire une trace d'exécution sous la forme d'une séquence d'événements sont plus adaptés au niveau RTL. Une argumentation détaillée sera fournie à la fin de ce chapitre (c.f. section 3.2.2.4).

Le modèle de surveillance

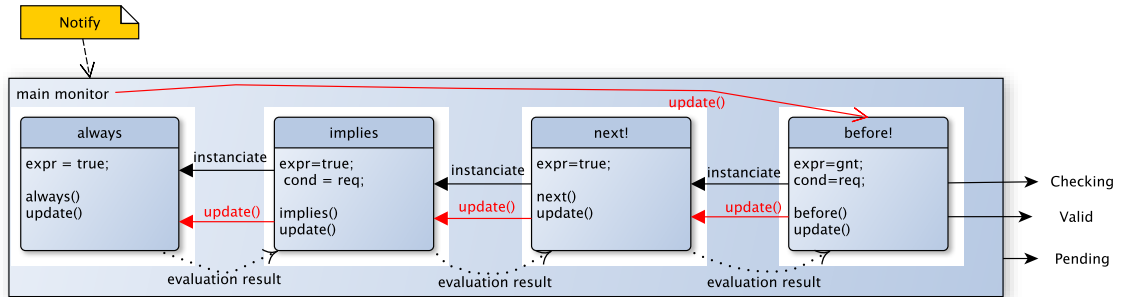
Le modèle de surveillance d'ISIS s'appuie sur un modèle bien connu de la programmation orientée objet : le motif “observateur-observable”[GHJV94]. L'objet observé (*Subject*) détient une liste des objets observateurs (*Observer*) et se charge de les aviser du changement de son état. Les observateurs, quant à eux, doivent spécifier la manière dont ils souhaitent réagir au changement d'état de l'objet observé. Dans le contexte d'ISIS chaque moniteur généré est placé dans un *wrapper*. Ce dernier se charge d'observer les composants du modèle à vérifier concernés par les transactions surveillées dans la propriété PSL. Le diagramme de classe UML de la Figure 3.15 illustre le modèle utilisé dans ISIS. La classe “Subject” définit l'interface des classes observées. La fonction `NotifyObs()` permet à l'objet de notifier les classes qui l'observent de son changement d'état. Dans le cadre de TLM, ce mécanisme est effectué à l'intérieur des fonctions de communication observées. La classe abstraite “*Observer*” définit l'interface à implémenter par tous les observateurs. Les observateurs sont les *wrappers*. La méthode `update()` doit contenir le traitement relatif à une notification de l'objet observé. Chaque *wrapper* peut contenir un ensemble de moniteurs. La classe “*Monitor*” est la classe mère de tous les moniteurs élémentaires. Elle déclare la méthode d'évaluation de l'opérateur et de la propriété.

Le méthode d'évaluation

Nous reprenons l'exemple de l'assertion P_8 pour expliquer la méthode d'évaluation d'ISIS détaillée dans [Fer11].

FIGURE 3.15 – L’implémentation du paradigme *Observable-Observer* dans ISIS[Fer11]

Le moniteur global généré pour cette assertion est illustré par la Figure 3.16. Selon la figure, l’évaluation se fait en enchaînant les appels à la méthode `update()` des moniteurs élémentaires. Le moniteur le plus à droite dans la formule PSL est le premier à recevoir la notification d’activation envoyée par le *wrapper*. Avant de mettre à jour la valeur de ses sorties, ce moniteur fait suivre l’appel au moniteur qui le précède, ici le **next**. L’enchaînement se suit jusqu’à atteindre le premier moniteur élémentaire de la formule. De cette manière, le résultat d’évaluation des moniteurs est propagé dans le sens inverse d’appel à la méthode `update()` jusqu’au dernier moniteur élémentaire.

FIGURE 3.16 – Moniteur global généré par ISIS pour P_8

La spécification des propriétés PSL

La bibliothèque des moniteurs élémentaires implémentée pour ISIS contient tous les opérateurs FL temporels. La sémantique implémentée est conforme à la sémantique formelle de PSL_{ss} . Afin de concrétiser la spécification des propriétés PSL dans ISIS, nous proposons de reprendre l’exemple de la plateforme `pv_dma` décrite au deuxième chapitre (*c.f.* section 2.2.1.3). Nous suggérons de vérifier le protocole de programmation du DMA donné par la spécification textuelle suivante : “à chaque fois que le CPU demande le démarrage d’un transfert (i.e. écrit la valeur `START` dans le registre de contrôle

du DMA) alors il doit attendre la fin du transfert courant (i.e. notification par irq) avant de demander le démarrage d'un nouveau transfert". L'assertion PSL P_9 qui formule cette propriété est donnée dans le Programme 5.1.

Les auteurs d'ISIS tirent avantage du fait que la couche booléenne de PSL s'appuie sur le langage de modélisation sous-jacent pour exprimer sous forme booléenne les conditions sur les appels de fonctions. C'est un choix astucieux qui permet d'exprimer plus naturellement les propriétés TLM sans passer par des variables fictives.

```
vunit P9 {
    // HDL_DECLs :
    const int CTRL_REG = 12+16384; //the DMA control register
    const int START = 1; // the START value

    // HDL_STMTs :

    // PROPERTY :
    assert
    always ( (cpu_initport.write_CALL() &&
              cpu_initport.write.p1 == CTRL_REG &&
              cpu_initport.write.p2 == START)
            ->next(dma_irq.write_CALL() &&
                  dma_irq.write.p1 == true
                  before! (cpu_initport.write_CALL() &&
                           cpu_initport.write.p1 == CTRL_REG &&
                           cpu_initport.write.p2 == START)));
}
```

PROGRAMME 3.1 – La propriété P9 pour le pv_dma

L'outil génère automatiquement des méthodes ad-hoc ayant une valeur de retour booléenne, qui incluent les fonctions de notification nécessaires pour notifier l'appel (début) et le retour (fin) de la fonction. Pour cette raison, pour chaque méthode observée, ISIS génère deux méthodes booléennes en ajoutant les suffixes “_CALL”, et “_END” au nom original. Ainsi, l'expression booléennes `cpu_initport.write_CALL()` dans le texte de la propriété ci-dessus signifie que l'utilisateur veut observer les appels à la fonction `write` au niveau du port initiator du CPU. ISIS fournit aussi la possibilité d'exprimer des conditions sur les valeurs des paramètres des fonctions observées. Syntaxiquement, cela s'obtient par le motif `composant.fonction.p#` où # est la position du paramètre dans la fonction (0 pour la valeur de retour) ou encore en spécifiant le nom du paramètre suivant le motif `composant.fonction.parametre`. L'utilisation du suffixe “_END” est particulièrement utile pour permettre la récupération de la valeur de retour disponible à la fin d'exécution de la fonction.

Concernant les signaux booléens, l'observation ne peut se faire que sur les appels à

la fonction `write` de la classe `sc_signal` (c.f. page 28). ISIS offre une simplification syntaxique en considérant que l'expression `dma_irq.write_CALL() && dma_irq.write.p1 == true` est équivalente à l'expression `dma_irq`. Dans les prochains chapitres, nous considérons que l'assertion utilise la version réduite de cette expression.

Par défaut, l'ensemble des fonctions observées spécifie l'échantillonnage utilisé. Comme indiqué dans la section 3.1.2, PSL prévoit l'opérateur `@` pour définir les expressions booléennes selon lesquelles la trace de l'assertion va être échantillonnée. Dans le cas d'ISIS, l'utilisateur peut aussi utiliser cet opérateur afin de choisir explicitement les méthodes booléennes qui vont déterminer l'échantillonnage. La propriété P_9 contient implicitement l'expression `@(cpu_initport.write_CALL() || dma_irq.write_CALL())`.

Dans la propriété énoncée, les parties délimitées par les commentaires “HDL_DECLs” et “HDL_STMTs” appartiennent à la couche modélisation de PSL. Dans PSL, cette couche permet d'introduire des constantes et des variables auxiliaires utilisées pour le stockage de certaines valeurs. Ces variables peuvent être modifiées plusieurs fois durant la vérification de la propriété. Dans le cas de la propriété P_9 , cette couche n'est utilisée que pour définir des constantes.

Dans [PSL05], la définition de la couche modélisation se limite à une caractérisation syntaxique. ISIS propose une extension de la sémantique de PSL afin d'introduire un support pour les variables globales de la couche modélisation. Cette extension est basée sur les travaux de [CM04] définissant un ensemble de règles de sémantique opérationnelle. Dans [Fer11], toutes les règles de la sémantique opérationnelle, à l'exception de celles des SERES, ont été reprises pour inclure la prise en considération des variables globales. D'un point de vue syntaxique, la couche modélisation est divisée en deux parties : une partie déclarative (`// HDL_DECLs`) et une partie pour le bloc d'instructions définissant l'évolution des variables déclarées (`// HDL_STMTs`).

```
vunit P10 {
    // HDL_DECLs :
    const int SRC_REG = 0x4000 + pv_dma::SRC_ADDR;
    const int M1_BOUND = 0x100;
    // Declaration de la variable auxiliaire :
    int req_src_addr;
    // HDL_STMTs :
    // Memorisation de la donnée écrite dans le registre du DMA :
    if(cpu_initport.write_CALL() && cpu_initport.write.p1 == SRC_REG)
        req_src_addr = cpu_initport.write.p2;
    // PROPERTY :
    assert
    always ( (cpu_initport.write_CALL() &&
```

```

    cpu_initport.write.p1 == SRC_REG &&
    cpu_initport.write.p2 < M1_BOUND)
->next_event!(memory1.read_CALL())
    (memory1.read.p1 == req_src_addr) );
}

```

PROGRAMME 3.2 – La propriété P10 pour le pv_dma

La propriété P_{10} donne un exemple de l'utilisation de la couche modélisation selon la sémantique et la syntaxe définies dans ISIS. Cette propriété spécifie que : “ à chaque fois que le CPU programme le DMA pour effectuer un transfert depuis une adresse source (i.e. écrit une adresse dans le registre `SRC_REG` du DMA) alors la prochaine lecture dans cette mémoire sera effectuée à cette adresse”. Dans cet exemple, la couche modélisation est utilisée pour sauvegarder la valeur de l'adresse écrite dans le registre `SRC_REG` du DMA. Le bloc d'instructions de la couche modélisation contient une instruction conditionnelle qui permet d'indiquer que si l'opération à l'origine de la notification du moniteur de la propriété est une opération d'écriture à destination du registre source du DMA, alors la valeur écrite sera mémorisée dans la variable auxiliaire `req_src_addr`. Ce bloc sera exécuté juste avant chaque évaluation de l'assertion.

La relation entre les identifiants et les composants

Le lien entre les identifiants utilisés dans l'assertion et les composants effectifs du modèle à surveiller est assuré par un fichier particulier appelé “fichier d'instrumentation” fourni par l'utilisateur. Le programme 3.3 montre le fichier d'instrumentation associé à la propriété P_9 . Il s'agit d'un fichier XML²⁰ composé d'un ensemble de liens (*links*) dont chacun établit la correspondance entre un composant du modèle et la variable utilisée dans la propriété PSL. Dans l'exemple, l'expression `cpu/initiator_port` dans la balise `<component>` indique, par convention, le port *initiator* du CPU.

²⁰eXtensible Markup Language : un standard du consortium *World Wide Web*. C'est un format de stockage qui permet une représentation structurée et extensible des informations textuelles. <http://www.w3schools.com/xml/default.asp>

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE instrumentation SYSTEM "instrumentation_dtd.dtd">
<instrumentation>
  <property>P9</property>
  <links>
    <link>
      <component>cpu/initiator_port</component>
      <variable>cpu_initport</variable>
    </link>
    <link>
      <component>pv_dma_irq</component>
      <variable>dma_irq</variable>
    </link>
  </links>
</instrumentation>

```

PROGRAMME 3.3 – Fichier d'instrumentation de P_9

3.2.2.4 L'utilisation des SERES et de l'opérateur next

Comme le démontre l'exemple de la SERE P_2 donné dans la Figure 3.3, les SERES permettent de décrire une séquence précise de signaux sur une succession finie de cycles d'horloge. Au niveau RTL, les assertions PSL contiennent couramment l'opérateur `next`, ainsi que les formules à base de SERES. Cette utilisation est justifiée à ce niveau car l'échantillonnage est fixe. Considérons, par exemple, un système synchrone sur les fronts montants de l'horloge clk décrit au niveau RTL et comportant quatre signaux a , b , c , d et une assertion A impliquant deux signaux a et d qui indique que d doit avoir lieu quatre cycles après la production de a . Cette propriété peut être spécifiée en PSL en utilisant la formule suivante : $a \rightarrow \text{next}[4](d)$. La Figure 3.17 illustre un extrait de trace d'exécution du système. La première lettre de cette trace contient a , et la cinquième contient d . L'assertion A est donc satisfaite sur cette trace obtenue en échantillonnant sur les fronts montants de clk .

Pour les assertions transactionnelles, l'utilisation de ces opérateurs doit se faire avec précaution. La plupart des modèles transactionnels ne contiennent pas d'horloge permettant un échantillonnage fixe de la trace ; tel est le cas pour le `pv_dma`. Dans ce cas, l'échantillonnage de la trace des assertions se fait relativement aux événements considérées dans ces assertions. Par conséquent, la "distance" entre les événements dans la trace peut différer en fonction de l'échantillonnage. Nous proposons de faire appel à l'exemple du DMA, décrit dans la section 2.2.1.3, pour donner une explication intuitive. La trace d'exécution de la Figure 3.18 montre la suite des transactions constituant le scénario de transfert d'un bloc de données. Le scénario considéré commence par l'écriture faite par le CPU dans le registre de contrôle du DMA permettant de démarrer le transfert. L'évène-

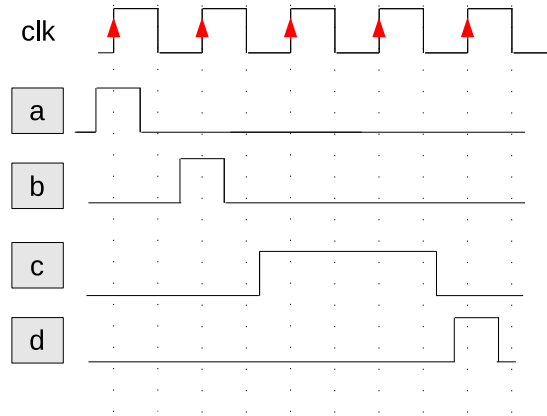


FIGURE 3.17 – Extrait de trace d'exécution du système RTL

ment associé est marqué par le point (1). Cette opération est suivie par une succession de “nb_word” transferts dont chacun est constitué d’une opération de lecture suivie d’une opération d’écriture. Ces opérations constituent les points allant de (2) à (n+1). La fin du scénario est marquée par l’envoi de l’interruption par le DMA. Ce point est noté (n+2).

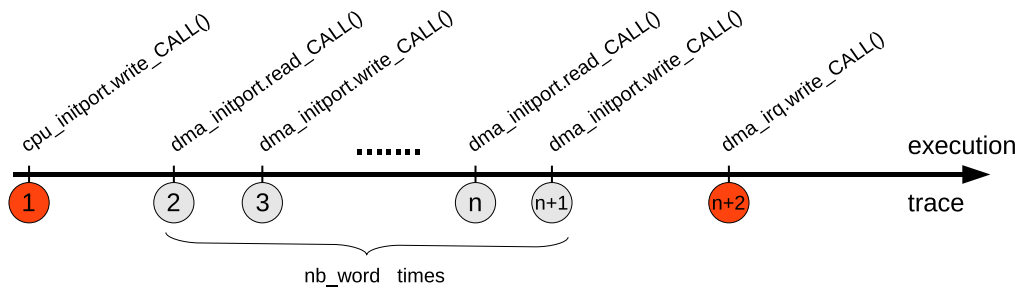


FIGURE 3.18 – Scénario d'exécution du transfert d'un bloc de données

Considérons maintenant qu’on souhaite surveiller la propriété indiquant que chaque transfert programmé par le CPU doit être suivi par une interruption. Cette propriété ne porte que sur les points (1) et (n+2). Ainsi, seuls ces deux événements sont pris en compte pour la constitution de la trace de la propriété afin d’éviter le sur-échantillonnage de la trace par des informations parasites. Sur la sous trace composée de ces deux points, nous pourrions spécifier l’assertion suivante afin de surveiller le comportement indiqué : `always (cpu_initport.write_CALL() ->next(dma_irq))`. Cette assertion est satisfaite par la sous trace mais elle serait violée dès qu’on prendrait en considération l’un des autres événements de la trace d’exécution. Le raisonnement est le même pour les opérateurs SERE. Une autre cause auxiliaire est liée à la nature non-déterministe du moteur de simulation SystemC qui rend difficile la prédiction de l’ordre exact d’activation des processus.

En conclusion, au niveau transactionnel, l’emploi de ces opérateurs qui indiquent des décalages exacts dans la trace n’est pas pertinent. Il paraît plus judicieux de faire appel

aux opérateurs temporels qui font référence à des relations et d'ordre tels que `until`, `before`, `next_event`, etc. C'est la raison pour laquelle une formalisation adaptée est celle de la propriété P_9 , qui spécifie seulement que `dma_irq` doit se produire à n'importe quel instant mais avant l'écriture suivante du CPU dans le registre de contrôle du DMA.

Toutefois, il faut souligner que l'opérateur `next` peut être utile pour exclure l'instant d'évaluation courant dans une sous-formule PSL. C'est le cas par exemple pour la propriété "`a -> (b before a)`". Cette propriété sera violée dès la première occurrence de l'expression `a` car `a` est déjà vrai dans la sous formule "`(b before a)`". En revanche, en ajoutant l'opérateur `next` comme suit : "`a ->next(b before a)`", la première occurrence de l'expression `a` permettra de démarrer l'évaluation de la sous formule à l'instant suivant. On remarque que c'est d'ailleurs le cas pour P_9 .

3.3 Bilan

Bien que la vérification des systèmes complexes consomme la plus grande partie de l'activité globale, peu d'outils sont proposés pour la vérification au niveau système. L'absence d'un flot de vérification conjoint au flot de conception freine considérablement les efforts de vérification au niveau système.

Nous avons montré au cours de ce chapitre que des travaux antérieurs ont permis de produire des solutions efficaces pour la vérification au niveau TLM et au niveau RTL. Nous rappelons en particulier respectivement ISIS et Horus. Il apparaît toutefois que les assertions temporelles mises en jeu se situent à des niveaux de granularité temporels et structurels très différents puisqu'une grande quantité de détails sera introduite durant le raffinement conceptuel vers RTL. Pour assurer une continuité du flot de vérification, une méthode permettant de décliner et vérifier au niveau RTL les propriétés vérifiées sur le modèle de référence au niveau TLM sera nécessaire. Elle fait l'objet du chapitre suivant.

Le raffinement des assertions

Sommaire

Introduction	108
5.1 Outil de raffinement des propriétés PSL	109
5.1.1 Les étapes de pré-raffinement	111
5.1.2 Le processus de raffinement	111
5.1.3 Réutilisation des fichiers Excel	121
5.2 Extensions d’ISIS	122
5.2.1 Prise en compte des fronts montants ou descendants de signaux autres que des signaux de type horloge	122
5.2.2 Possibilité de consulter les valeurs des signaux	122
5.3 L’implémentation des SEREs	124
5.3.1 Définition des moniteurs SERE élémentaires	124
5.3.1.1 Moniteurs SERE dans Horus	124
5.3.1.2 Moniteurs SERE dans ISIS	126
5.3.2 Méthode d’interconnexion des SEREs	132
5.3.2.1 Interconnexion des moniteurs FL élémentaires	132
5.3.2.2 Interconnexion des moniteurs SERE élémentaires	132
Bilan	134

*“The process of creating assertions forces the engineer to think. . .
and in this incredible world of automation, there is no substitute for thinking”*

Harry D. Foster -Assertion Based Verification- 2010

Introduction

Dans ce chapitre, nous allons tout d’abord discuter les enjeux et les objectifs du raffinement des assertions entre les niveaux transactionnels et RTL. Un tour d’horizon des différents travaux réalisés dans ce contexte sera présenté et commenté par la suite, suivi d’une explication détaillée de l’approche de raffinement proposée.

4.1 Les motivations et enjeux du raffinement

Les problématiques liées au raffinement des assertions sont principalement issues de la disparité des domaines transactionnels et RTL. Les différences peuvent être identifiées en deux catégories : structurelles et temporelles.

4.1.1 Les différences structurelles

Les assertions énoncées au niveau transactionnel concernent principalement les communications entre les composants. L’analyse des différences structurelles à prendre en compte dans le processus de raffinement se concentre alors sur les interfaces de communication.

Au niveau transactionnel, le niveau de granularité des communications est la transaction. On parle d’une interface de communication basée sur les transactions. Les transactions sont réalisées par des appels de fonctions de transport abstraites bloquantes ou non bloquantes. Les données transportées sont passées en paramètres aux fonctions de transport ou récupérées en valeur de retour, et les types de données peuvent être des types de haut niveau comme des entiers, voire même des classes C++.

De l’autre côté, au niveau RTL, les communications sont réalisées par les signaux reliant les ports des composants. Les types des signaux sont *bit-accurate* (c.f. section 2.1). Les interfaces RTL sont *pin-accurate* (c.f. section 2.1) et sont souvent conformes à un protocole de communication standard ou propriétaire. La Figure 4.1 illustre deux exemples de raffinement de l’interface d’un maître (*master*) et d’un esclave (*slave*) du niveau TLM vers les interfaces d’un bus Wishbone (a) et AMBA-AHB (b) au niveau RTL. On remarque alors que le raffinement structurel des interfaces de communication dépend du protocole du modèle RTL sous-jacent et du rôle du composant dans les communications. Nous notons que par exemple, selon la Figure 4.1, 15 signaux sont nécessaires pour implé-

menter le protocole AHB tandis que 10 sont suffisants pour l'implémentation du protocole Wishbone classique. Le protocole AHB possède par exemple, en plus, des signaux dédiés à l'arbitrage (HBUSREQ, HLOCK) lui permettant de demander l'accès au bus. L'interface standard du Wishbone peut en revanche être étendue pour ajouter des signaux spécifiques aux transferts rafales tels que CTI_I/O et BTE_I/O.

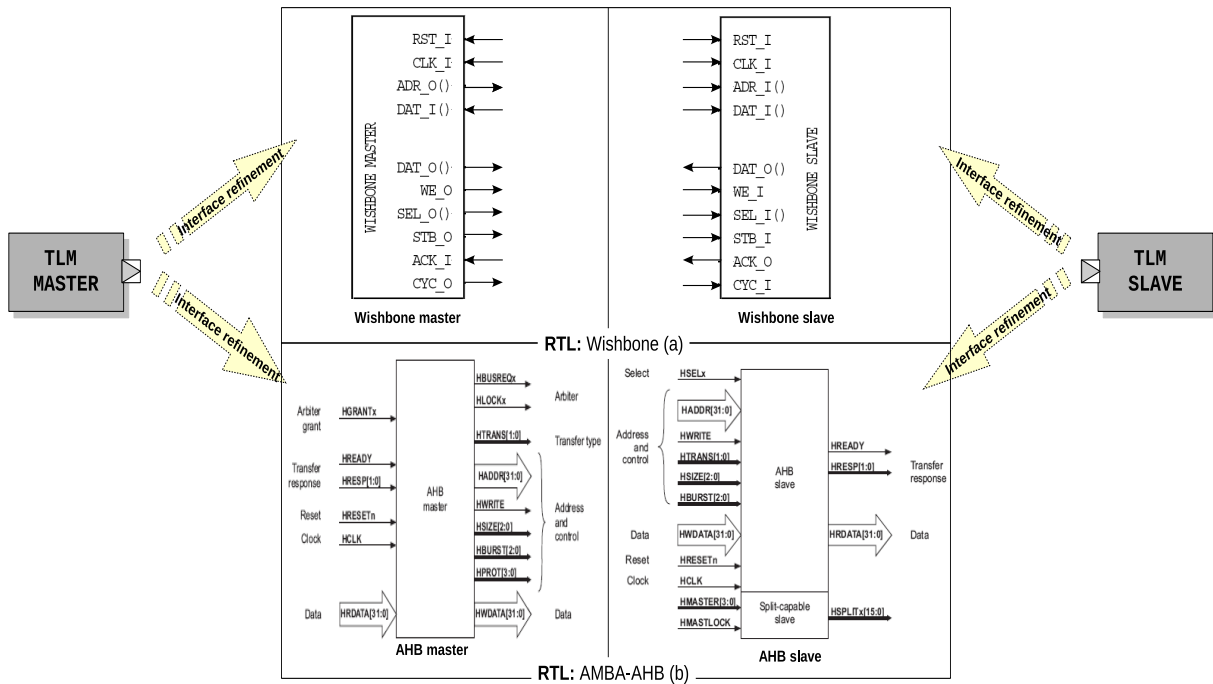


FIGURE 4.1 – Raffinement des interfaces des composants maître et esclave TLM

Par conséquent, lors du raffinement structurel, les correspondances structurelles entre les transactions et les signaux doivent être faites ainsi que la correspondance entre les composants TLM et les composants RTL. Concrètement, il est nécessaire d'établir la correspondance entre les fonctions de haut niveau et leurs paramètres d'un côté et les signaux RTL de l'autre côté.

4.1.2 Les différences temporelles

La principale difficulté dans le processus de raffinement est la variation du niveau de granularité temporelle entre les niveaux transactionnels et le niveau RTL. Au niveau le plus abstrait, les transactions sont des opérations atomiques, éventuellement même réalisées en un temps nul.

Au niveau LT, les fonctions de transport sont typiquement bloquantes et peuvent contenir des délais d'attente. Cependant, une transaction de lecture ou d'écriture nécessite un seul appel à la fonction de transport bloquant (*c.f.* section 2.2.2.1).

Au niveau AT, en revanche, les fonctions de transport peuvent être amenées à être non-bloquantes. Une transaction est composée de plusieurs phases identifiant des points

temporels différents. Deux fonctions de transport sont nécessaires pour réaliser une transaction et d'autres appels aux fonctions de transport non bloquantes peuvent avoir lieu entre le début et la fin de cette transaction.

Dans tous ces cas, l'identification des transactions se fait en se référant aux événements de début et de fin de ces transactions. Ces événements sont atomiques indépendamment du niveau d'abstraction transactionnel. L'échantillonnage de la trace d'exécution se fait sur la base de ces événements. La Figure 4.2 illustre un exemple d'identification des transactions résultant des appels à une fonction d'écriture bloquante (`b_write`) et non bloquante (`nb_write`). Les événements de début et de fin des transactions sont dénotés respectivement par l'ajout des postfixes “_START” et “_END”.

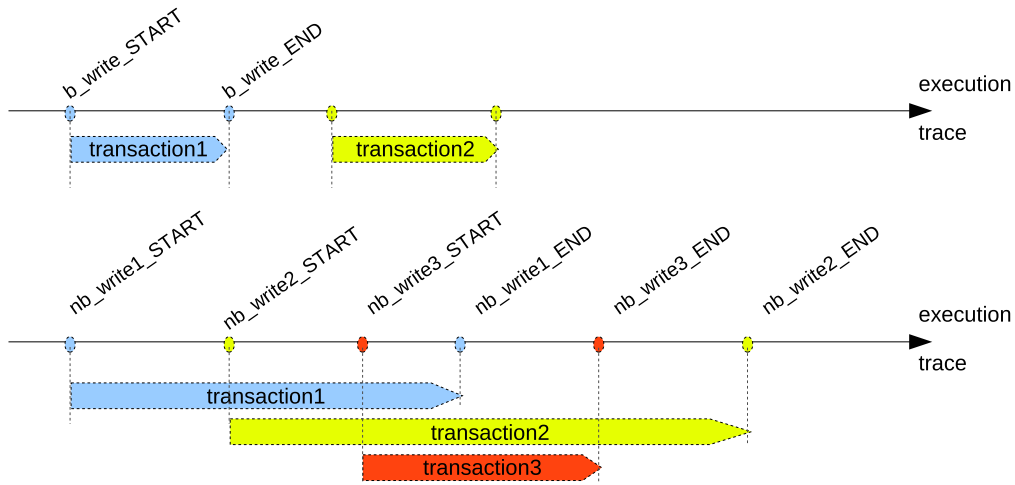


FIGURE 4.2 – Exemples d'appels à des fonctions bloquantes et non bloquantes

De l'autre côté du flot, au niveau RTL, la précision temporelle est plus importante grâce à l'introduction d'un protocole de communication bien défini. Pour les protocoles de communication synchrones, les opérations de lecture et d'écriture sont généralement modélisées par un ensemble de signaux de contrôle, d'adresse et de données dont le comportement est spécifié cycle par cycle pour chaque type de transfert implémenté.

À titre d'exemple, la Figure 4.3 décrit le déroulement d'une opération d'écriture simple selon le protocole Wishbone dans (a) et le protocole AMBA-AHB dans (b). On remarque alors que le comportement temporel des signaux peut changer selon le protocole implémenté et ce pour le même type de transfert. Dans le cas du bus Wishbone, un seul cycle d'horloge est nécessaire pour réaliser toute l'opération d'écriture simple. Dans le cas du protocole AMBA-AHB, deux cycles sont nécessaires pour réaliser cette même opération. Le premier cycle correspond à la phase d'adresse dans laquelle les signaux de contrôle sont positionnés et l'adresse est envoyée sur le bus d'adresse “`HADDR`”. Le second cycle correspond à la phase de données durant laquelle les données sont placées sur le bus d'écriture des données “`HWDATA`”.

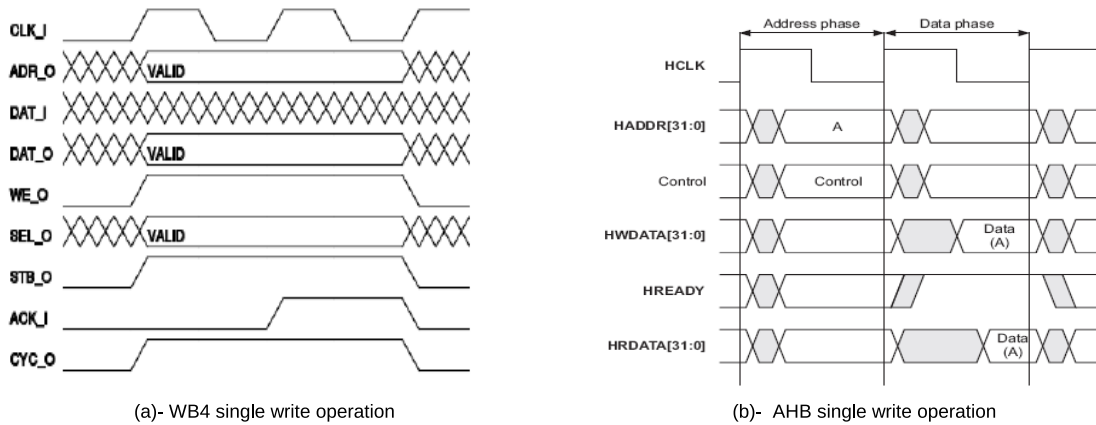


FIGURE 4.3 – Les chronogrammes d’une écriture simple pour le bus wishbone (source :[ope10]) et AHB (source :[ARM08])

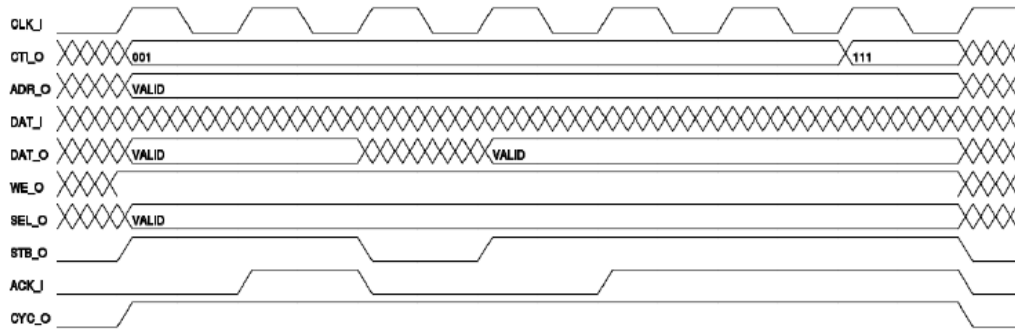


FIGURE 4.4 – Une opération d’écriture *burst* du bus Wishbone (source :[ope10])

La Figure 4.4 illustre une opération d’écriture bloquante de type rafale ou *burst*¹ selon le protocole Wishbone. Dans ce cas, l’opération est réalisée sur plusieurs cycles afin de permettre l’écriture d’un bloc entier de données étalées sur des adresses adjacentes. Le signal `CTI_0` permet d’identifier le type du transfert. Dans le cas représenté par cette figure, il s’agit d’une rafale à une adresse constante (une FIFO par exemple). Le transfert commence quand `CYC_0` et `STB_0` sont au niveau haut. La fin de la rafale est déterminée par l’envoi d’une séquence de bits particulière (111) via le port `CTI_0`². À chaque fois que le signal `STB_0` descend au niveau bas, le transfert est suspendu.

¹c’est une transmission répétitive de données sans passer par toutes les étapes de transmission nécessaires pour une transmission simple d’une seule donnée. Le but de ce transfert est d’augmenter le débit de transmission des données sur le bus. Il est alors souvent utilisé pour l’accès aux supports de stockage de données (mémoires, disques dur) nécessitant un long temps d’accès.

²Dans le cas du transfert simple représenté par la Figure 4.3(a), ce signal n’a pas été représenté car il a gardé sa valeur par défaut.

4.1.3 L'exemple du pv_dma

Si on reprend l'exemple du pv_dma présenté dans 2.2.1.3, l'opération d'écriture atomique au niveau transactionnel peut s'étaler au niveau RTL sur un ou plusieurs cycles d'horloge selon le type du transfert et le protocole implémenté. Par exemple, pour la configuration des registres du DMA par le CPU réalisée par un appel à la fonction `write(const ADDRESS& address, const DATA& data)`, des transferts simples sont suffisants.

Selon la spécification illustrée dans la Figure 4.3(b), ce transfert simple dure deux cycles d'horloge dans le cas du protocole AHB. L'opération d'écriture atomique au niveau transactionnel se traduit alors par un transfert sur deux cycles d'horloge comme le montre la Figure 4.5.

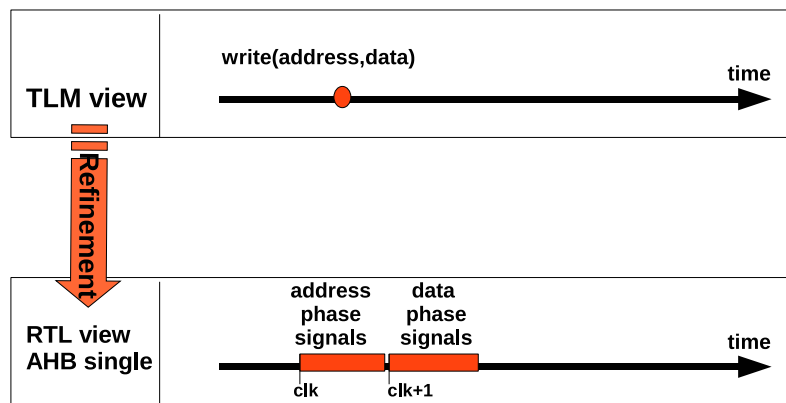
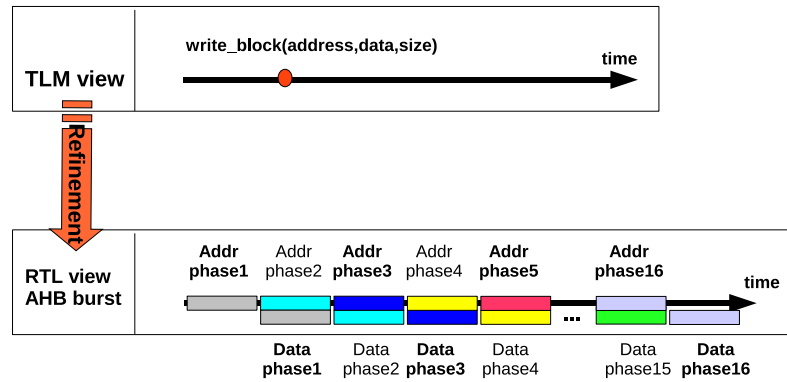


FIGURE 4.5 – Le raffinement temporel d'une opération d'écriture simple selon le bus AHB

Cependant, durant l'écriture dans la mémoire, le DMA peut aussi faire appel à la méthode `write_block(const ADDRESS& address, const DATA * data, const unsigned int block_size)` pour permettre une écriture atomique d'un bloc de 16 mots dans la mémoire (*c.f.* programme 2.6). Au niveau RTL, si le protocole AHB est implémenté, un transfert en rafale est nécessaire pour réaliser cette opération. La fonction d'écriture atomique “`write_block`” se traduit dans ce cas par un ensemble de transferts s'étalant sur plusieurs cycles d'horloge (`=nb_word`). Comme le montre la Figure 4.6, à chaque cycle i , $1 < i < \text{nb_word}$, les signaux de contrôle et d'adresse du transfert i sont configurés et les données du transfert $i - 1$ sont envoyées.

En conclusion, lors du raffinement temporel des assertions, il est nécessaire d'identifier et de prendre en considération les contraintes structurelles et temporelles imposées par les spécifications des protocoles de communication implémentés dans les modèles *cycle-accurate*.

Il faut également identifier l'horloge de synchronisation utilisée au niveau RTL.

FIGURE 4.6 – Le raffinement temporel d’une opération d’écriture *burst* selon le bus AHB

4.2 Travaux sur le raffinement des assertions

Très peu de solutions ont été proposées pour le raffinement d’assertions temporelles. Dans cette section, nous proposons de citer et commenter les principaux travaux de recherche. Nous nous pencherons en particulier sur les travaux de Fummi et al. [BFF05] [BFP07] [BDF08] et Ecker et al. [EEH⁺06] [Ese08] [Ste12].

4.2.1 Approches à base de transacteurs

Dans [BFF05] [BFP07] et [BDF08], une approche de création d’un flot de vérification à base de transacteurs a été développée. Le point d’entrée de ce flot est un ensemble d’assertions transactionnelles. Il s’agit d’assertions purement booléennes servant généralement à exprimer des relations entre les entrées et les sorties primaires. La vérification des relations temporelles et des communications ne sont pas prises en considération.

L’application des assertions transactionnelles suit le schéma décrit par la Figure 4.7 :

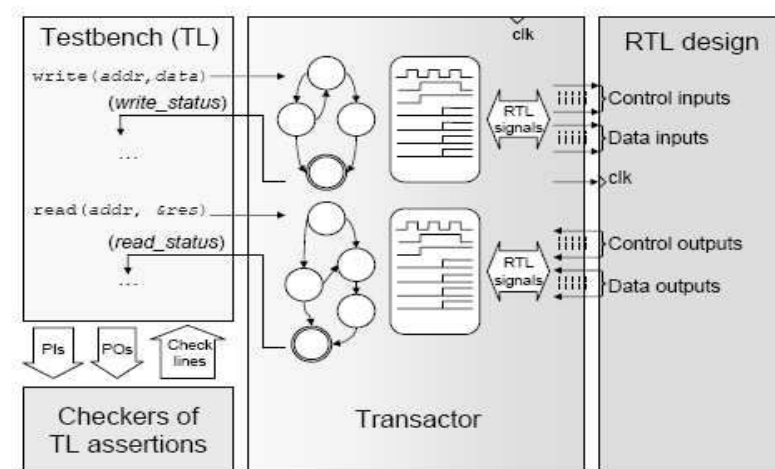


FIGURE 4.7 – Méthode de réutilisation des assertions transactionnelles (source : [BFP07])

La méthode consiste à appliquer un *testbench* composé d’une opération d’écriture

“`write(addr, data)`” suivie d’une opération de lecture “`read(addr, &res)`” permettant de vérifier l’opération d’écriture. Le rôle des transacteurs est de traduire ce vecteur de test en un ensemble de séquences de test caractérisant le protocole de communication implémenté au niveau RTL. L’état de chaque opération (`write_status`, `read_status`) appliquée est retourné au *testbench* afin de s’assurer de l’absence d’erreurs. À la fin de l’exécution du vecteur de test, les paramètres des fonctions (`addr`, `data`, `write_status`, `&res`, `read_status`) sont fournis en entrée des moniteurs transactionnels.

Outre cette solution à base de transacteurs, les auteurs mettent en œuvre des assertions fournies avec les descriptions industrielles de bus de communication, ou testent que les résultats calculés pour les composants arithmétiques RTL sont identiques aux résultats fournis par les fonctions C associées (par exemple `sqrt(n)`).

Dans [KT07], une approche similaire a été brièvement décrite permettant de vérifier les modèles raffinés au niveau *cycle-accurate* par les assertions transactionnelles. La solution consiste à développer des transacteurs à différents niveaux d’abstraction et à les lier au testbench au niveau transactionnel.

Dans les deux travaux cités, le principal inconvénient est que les assertions transactionnelles traitées appartiennent toutes à la couche booléenne. Il n’est alors pas possible de spécifier des comportements temporels. L’autre inconvénient de ces solutions est relié à l’utilisation des transacteurs. En effet, ces composants fonctionnent comme de simple adaptateurs entre des appels de fonctions transactionnels et un ensemble de signaux au niveau RTL et ne fournissent pas une caractérisation sémantique du raffinement des assertions transactionnelles. Contrairement à notre approche, le raffinement ne génère pas en sortie une assertion formelle susceptible d’être utilisée par un outil de vérification formelle.

4.2.2 Approche de Ecker et al.

Dans [EESV07] [Ste12], **Ecker, Steininger et al.** se réfèrent au premier standard OSCI TLM-1[OSC05] pour considérer trois niveaux d’abstraction transactionnels : PV (Programmer’s View), PVT (Programmer’s View with Timing) et CA (*cycle-accurate*).

Au niveau PV non-temporisé, la synchronisation est faite uniquement sur les occurrences discrètes des événements de début et de fin des transactions. Par définition, cette relation est appelée “*Event Based Simultaneity*”.

Au niveau PVT, le temps de simulation n’est pas nul. Les événements des transactions peuvent alors être associés à une certaine valeur temporelle correspondant au temps de simulation, cette relation est dite “*Time Based Simultaneity*”.

Au niveau CA, la synchronisation est faite sur les cycles d’horloge. Il s’agit alors d’une relation appelée “*Cycle Based Simultaneity*”. Dans ces trois niveaux, les auteurs font appel au langage UAL pour la spécification des assertions (*c.f.* section 3.2.2.2). Ceci est possible grâce à la sémantique de l’opérateur de délai, définie indépendamment des couches

d'abstraction. Ceci permet notamment l'expression de séquences à différents niveaux d'abstraction ou encore regroupant plusieurs niveaux d'abstraction (hybrides).

4.2.2.1 La transformation des assertions

Le langage UAL a été initialement conçu pour la vérification des modèles transactionnels. Dans [Ste12], une extension du langage dite UAL+ a été définie pour couvrir en plus le niveau RTL en vue de l'établissement d'un processus de transformation des propriétés transactionnelles vers leur équivalent RTL et vice-versa, qui ne concerne que la couche booléenne. Par la suite, un nouveau langage de transformation des assertions a été défini.

Comme le montre la Figure 4.8, le fichier décrivant les transformations définies par l'utilisateur doit être fourni en entrée du processus de transformation (*Refinement Generator*). Ce processus se déroule en deux étapes : l'exécution des règles de transformation permet la transformation temporelle alors que l'instanciation des transacteurs permet la transformation structurelle. Il génère en sortie l'assertion transformée.

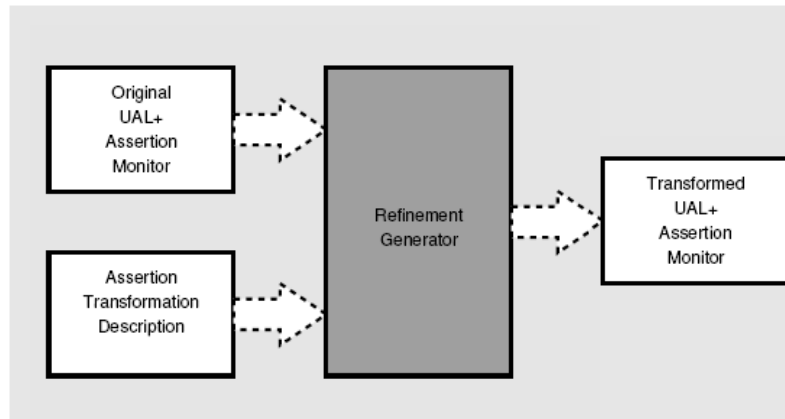


FIGURE 4.8 – Processus de transformation selon Ecker et al. (source : [Ste12])

La transformation de l'assertion est réalisée en exécutant les directives du fichier de transformation directement sur l'arbre de l'assertion d'origine. L'assertion résultante est générée à partir de l'arbre modifié. La transformation des assertions est possible dans les deux sens : en abstraction (de RTL vers PV) ou en raffinement (de PV vers RTL). Contrairement au processus d'abstraction, le processus de raffinement ne peut pas se faire directement entre les niveaux PV et RTL, un raffinement en deux étapes, en passant par le niveau PVT, est nécessaire pour effectuer une première transformation temporelle. Entre le niveau CA et RTL, la transformation est purement structurelle.

Nous soulignons que dans les deux cas, les transformations appliquées se limitent à remplacer une expression de la couche booléenne (ou de la couche événement) par une autre expression du même type. Cela vise, dans le passage du niveau PV au niveau PVT à remplacer par exemple une expression du style “une interruption I se produit” par une

expression du style “une interruption I se produit dans les K ns”, permettant ainsi de refléter dans la condition le délai explicité dans la description PV+*timing*. Le passage au niveau CA conduira à traduire un délai temporel en un nombre de cycles d’horloge correspondant. L’exemple ci-dessus devient “une interruption I se produit dans les N cycles d’horloge”, si N cycles correspondent à K ns.

Lors de la transformation, les auteurs exigent donc que la propriété préserve sa structure syntaxique en dépit de l’ajout des informations relatives au raffinement du modèle. Autrement, le résultat du raffinement est considéré erroné.

La préservation de la structure de l’assertion signifie que l’assertion transformée doit avoir le même nombre d’expressions booléennes, séquentielles et événementielles que l’assertion d’origine. Cette contrainte est incompatible avec notre analyse des exigences du raffinement temporel qui a démontré qu’en fonction du protocole de communication, une transaction s’exécutant sur un seul point temporel peut s’étaler sur plusieurs cycles au niveau RTL ce qui implique généralement la modification de la structure syntaxique de l’assertion, comme nous le verrons dans la section suivante (*c.f.* section 4.3).

4.2.2.2 Les restrictions

Selon [Ste12], le processus de transformation proposé comporte quelques restrictions telles que :

- Dans certains cas, devoir respecter la contrainte de préservation de la structure pourrait engendrer des erreurs dans le comportement des assertions générées. Dans l’exemple donné ci-dessus, la valeur temporelle spécifiée au niveau PVT doit correspondre exactement à un multiple de la période d’horloge au niveau CA. Dans le cas contraire, la transformation n’est pas possible.
- L’abstraction des assertions RTL n’est possible que si celles-ci respectent un format particulier spécifié par les auteurs. Cette représentation facilite le respect de la contrainte de préservation de la structure des assertions transformées.
- Le raffinement des assertions qui prennent en considération des transferts *pipelinés* n’est pas possible (Il en sera de même dans l’état actuel de notre méthodologie).
- Le mécanisme d’évaluation des assertions raffinées au niveau RTL peut être coûteux dans certains cas (présence de nombreux processus concurrents).

Globalement, les auteurs proposent un processus de transformation permettant à la fois le raffinement et l’abstraction des assertions en s’appuyant sur un langage de transformation bien défini. Toutefois, cette démarche exige, en contre partie, de l’utilisateur une connaissance poussée à la fois du langage de transformation et du langage de spécification.

4.3 Solution de raffinement

Notre objectif est de créer un flot de vérification basé sur le raffinement des assertions, et qui accompagne le flot de conception descendant tel qu'on l'a décrit dans la Figure 1.6. Idéalement, ce flot doit être applicable indépendamment des niveaux d'abstraction intermédiaires choisis. Afin de réaliser ceci, la méthodologie de raffinement des assertions doit être flexible et applicable aussi bien sur des composants entiers que sur une opération de communication d'un composant en particulier.

La méthodologie de raffinement des assertions transactionnelles proposée se déroule en deux étapes : le raffinement temporel des assertions permettant l'intégration des informations temporelles, et le raffinement structurel assurant une simple conversion des interfaces de communication transactionnelles vers le niveau *pin-accurate*. Le raffinement temporel est basé sur un ensemble de règles de transformations formelles (règles de réécriture) assurant la cohérence structurelle et sémantique de l'assertion résultante.

Dans les sections suivantes, nous considérons :

- φ , ϕ et ζ , trois formules PSL,
- $i, j, k, X \in \mathbb{N}$,
- B une expression booléenne,
- v une trace finie.

4.3.1 Premières règles de raffinement

Dans l'ensemble de règles de transformations $T = \{T_1, T_2, \dots, T_6\}$, chaque règle de transformation est une règle de réécriture $T_i : L \xrightarrow{C} R$ où L et R sont des modèles (patron) de formules. La transformation de L vers R se fait en fonction d'une contrainte temporelle C spécifiée selon le protocole de communication implémenté au niveau RTL. Par exemple, pour une opération d'écriture simple, la contrainte servira à indiquer, dans le cas du protocole AHB, que les données sont envoyées un cycle après l'envoi de l'adresse et des signaux de contrôle. L'ensemble T des règles de raffinement actuellement défini (présenté dans [PBHA13] et rappelé ici) permet d'automatiser le raffinement des communications en une séquence d'événements pour un certain nombre de transformations possibles. La transformation structurelle de l'assertion est faite après l'application des règles de transformation en remplaçant les expressions transactionnelles par les signaux correspondants.

4.3.1.1 Règles de transformation impliquant l'opérateur **before**

Transformation T_1

La première règle permet la transformation d'une expression temporelle PSL de la forme : “ φ **before**! (ψ and ζ)” dans le cas où le protocole de communication implémenté

au niveau RTL spécifie que **la condition ζ ait lieu X cycles d’horloge après la condition ψ** . Au niveau TLM, ces deux conditions seraient simultanées par rapport au même événement transactionnel d’où l’emploi de l’opérateur logique “and”. Typiquement, une telle relation peut être exprimée entre un appel de fonction de communication et une condition sur les paramètres de cette fonction comme dans le cas de la propriété P_9 extraite de l’exemple du pv_dma (*c.f.* programme 5.1). Dans cette propriété, au niveau de la formule :

```
 $F_1$  : dma_irq before! (cpu_initport.write_CALL() &&
cpu_initport.write.p1 == CTRL_REG &&
cpu_initport.write.p2 == START)
```

φ , ψ et ζ peuvent correspondre respectivement à :

- dma_irq,
- (cpu_initport.write_CALL() && cpu_initport.write.p1 == CTRL_REG),
- (cpu_initport.write.p2 == START).

La contrainte signifie que si ψ a lieu à l’instant courant (noté 0), alors ζ doit se produire X cycles après avec $X > 0$. Dans le cas du protocole AHB, par exemple, X sera égale à 1. Formellement, ceci peut s’exprimer par la contrainte “ $C : \psi$ **and next!** [X] (ζ)”.

“ φ **before!** (ψ and ζ)” deviendrait alors “ φ **before!** (ψ **and next!** [X] (ζ))”.

Selon les règles syntaxiques de PSL_{ss} , les deux opérandes d’un opérateur **before!** doivent être booléens (ne contiennent aucun opérateur temporel). Cette formule n’appartient pas alors au sous ensemble simple de PSL. Après la transformation formelle de cette formule vers le sous ensemble simple, on obtient la règle de transformation suivante (formule 4.3.1.1) :

$T_1 : \varphi \text{ before! } (\psi \text{ and } \zeta) \xrightarrow{C} ((\psi \rightarrow \text{next!} [X] (\neg \zeta)) \text{ until! } \varphi) \\ \text{and } \text{next.event!} (\varphi) (\psi \rightarrow \text{next!} [X] (\neg \zeta)),$ $C : \zeta \text{ occurs } X \text{ cycles after } \psi$
--

Si on reprend l’exemple de l’expression F_1 , la transformation de cette expression par l’application de la règle de transformation T_1 en intégrant la contrainte temporelle donnera en sortie la formule PSL citée dans 4.1 :

```

((cpu_initport.write_CALL() &&
  cpu_initport.write.p1 == CTRL_REG )
  -> next!(!(cpu_initport.write.p2 == START)))
    until! (dma_irq)
&&
next_event!(dma_irq)((cpu_initport.write_CALL() &&
  cpu_initport.write.p1 == CTRL_REG )
  -> next!(!(cpu_initport.write.p2 ==
    START)))

```

PROGRAMME 4.1 – Transformation de F_1 par la règle T_1

Transformation formelle vers PSL_{ss}

Nous allons donner, à titre d'exemple, la preuve formelle de la mutation de la règle initialement obtenue vers le sous ensemble simple PSL_{ss} . Proposons au préalable un petit rappel d'un ensemble d'équations de l'algèbre de Boole que nous utiliserons par la suite :

$$\neg(a \wedge b) \stackrel{\text{def}}{=} \neg a \vee \neg b \quad (4.1a)$$

$$\neg(a \vee b) \stackrel{\text{def}}{=} \neg a \wedge \neg b \quad (4.1b)$$

$$\neg a \vee b \stackrel{\text{def}}{=} a \rightarrow b \quad (4.1c)$$

$$a \vee (b \wedge \neg a) \stackrel{\text{def}}{=} a \vee b \quad (4.1d)$$

Comme nous l'avons mentionné dans 3.1.2.1, l'opérateur FL temporel **before**! est défini en fonction de l'opérateur **until**! selon la formule :

$$\varphi \text{ **before**! } \psi \stackrel{\text{def}}{=} (\neg\psi) \text{ **until**! } (\varphi \wedge \neg\psi)$$

La transformation de la formule vers PSL_{ss} commence alors par le remplacement de cet opérateur par sa définition :

$$v \models \varphi \text{ **before**! } (\psi \text{ and **next**!}[X]\zeta)$$

$$\text{i.e., } v \models \neg(\psi \text{ and **next**!}[X](\zeta)) \text{ **until**! } (\varphi \wedge \neg(\psi \text{ and **next**!}[X](\zeta)))$$

Selon la sémantique de **until**! donnée par l'équation 3.2a, on obtient :

$$\text{i.e., } \exists k < |v| \text{ s.t. } v^{k..} \models (\varphi \wedge \neg(\psi \text{ and **next**!}[X](\zeta))) \text{ and}$$

$$\forall j < k, v^{j..} \models \neg(\psi \text{ and **next**!}[X](\zeta))$$

D'une part, selon la logique formelle de PSL, on a :

$$v \models a \wedge b \stackrel{\text{def}}{=} v \models a \text{ and } v \models b \text{ [PSL05].}$$

Nous pouvons alors déduire aussi que :

$$v \models a \wedge b \Leftrightarrow v \models a \wedge b \text{ and } v \models a.$$

D'autre part, l'expression :

$$(\varphi \wedge \neg(\psi \text{ and } \mathbf{next}! [X]\zeta))$$

ne pouvant se produire qu'à partir de l'instant k , sa négation :

$$\neg(\varphi \wedge \neg(\psi \text{ and } \mathbf{next}! [X]\zeta)) \text{ est vraie avant cet instant.}$$

La formule devient :

$$\text{i.e., } \exists k < |v| \text{ s.t. } v^{k..} \models \varphi \text{ and } v^{k..} \models (\varphi \wedge \neg(\psi \text{ and } \mathbf{next}! [X]\zeta)) \text{ and}$$

$$\forall j < k, v^{j..} \models \neg(\varphi \wedge \neg(\psi \text{ and } \mathbf{next}! [X]\zeta)) \text{ and } v^{j..} \models \neg(\psi \text{ and } \mathbf{next}! [X]\zeta)$$

$$\text{i.e., } \exists k < |v| \text{ s.t. } v^{k..} \models \varphi \text{ and } v^{k..} \models (\varphi \wedge \neg(\psi \text{ and } \mathbf{next}! [X]\zeta)) \text{ and}$$

$$\forall j < k, v^{j..} \models \neg(\varphi \wedge \neg(\psi \text{ and } \mathbf{next}! [X]\zeta)) \wedge \neg(\psi \text{ and } \mathbf{next}! [X]\zeta)$$

Nous faisons appel à la formule 4.1(a) de la logique de Boole pour obtenir que :

$$\forall j < k, v^{j..} \models \neg(\varphi \wedge \neg(\psi \text{ and } \mathbf{next}! [X]\zeta)) \vee (\psi \text{ and } \mathbf{next}! [X]\zeta).$$

De la même manière, on fait appel à l'égalité 4.1(d) avec " $a = (\psi \text{ and } \mathbf{next}! [X]\zeta)$ " et " $b = \varphi$ " afin de déduire que :

$$\forall j < k, v^{j..} \models \neg(\varphi \vee (\psi \text{ and } \mathbf{next}! [X]\zeta))$$

Nous obtenons donc la formule suivante :

$$\text{i.e., } \exists k < |v| \text{ s.t. } v^{k..} \models (\varphi \wedge \neg(\psi \text{ and } \mathbf{next}! [X]\zeta)) \text{ and } v^{k..} \models \varphi \text{ and}$$

$$\forall j < k, v^{j..} \models \neg\varphi \text{ and } v^{j..} \models \neg(\psi \text{ and } \mathbf{next}! [X]\zeta)$$

Cette forme nous ramène vers deux définitions d'opérateurs FL de PSL :

- d'une part, $\exists k < |v| \text{ s.t. } v^{k..} \models \varphi \text{ and } \forall j < k, v^{j..} \models \neg(\psi \text{ and } \mathbf{next}! [X]\zeta)$ correspond à la définition de la sémantique de satisfaction de l'opérateur FL $\varphi_1 \mathbf{until}! \varphi_2$ avec $\varphi_1 = \neg(\psi \text{ and } \mathbf{next}! [X]\zeta)$ et $\varphi_2 = \varphi$.
- d'autre part, $\exists k < |v| \text{ s.t. } v^{k..} \models (\varphi \wedge \neg(\psi \text{ and } \mathbf{next}! [X]\zeta)) \text{ and } \forall j < k, v^{j..} \models \neg\varphi$ correspond à la sémantique de satisfaction de l'opérateur $\mathbf{next_event}! (B)$ (φ_1) définie en fonction de celle de l'opérateur $\mathbf{until}!$ (c.f. équation 3.8a), avec $B = \varphi$ et $\varphi_1 = \psi \text{ and } \mathbf{next}! [X]\zeta$.

En remplaçant les définitions sémantiques par les opérateurs FL correspondants, la formule devient :

$$\text{i.e., } v \models (\neg(\psi \wedge \mathbf{next}[X]!\zeta) \mathbf{until}! \varphi) \text{ and}$$

$$\mathbf{next_event!}(\varphi)(\neg(\psi \text{ and } \mathbf{next!}[X]\zeta))$$

i.e., $v \models ((\neg\psi \vee \neg(\mathbf{next}[X]!\zeta)) \mathbf{until!} \varphi) \text{ and } \mathbf{next_event!}(\varphi)(\neg\psi \vee \neg(\mathbf{next!}[X]\zeta))$ en utilisant la formule 4.1(a)

i.e., $v \models ((\neg\psi \vee (\mathbf{next}[X]!\neg\zeta)) \mathbf{until!} \varphi) \text{ and } \mathbf{next_event!}(\varphi)(\neg\psi \vee \mathbf{next!}[X]\neg\zeta))$ en utilisant la formule 4.1(a)

En appliquant la règle booléenne 4.1(c), avec $a = \psi$ et $b = \neg(\mathbf{next}[X]!\zeta)$, nous obtenons au final la partie droite R de la règle de transformation T_1 :

i.e., $v \models ((\psi \rightarrow \mathbf{next}[X]!\neg\zeta)) \mathbf{until!} \varphi \text{ and } \mathbf{next_event!}(\varphi)(\psi \rightarrow \mathbf{next}[X]!\neg\zeta))$

Transformation T_2

La règle de transformation T_2 s'applique sur les formules de la forme :

$$L : (\varphi \text{ and } \psi) \mathbf{before!} \zeta.$$

Dans cette règle, l'insertion de la contrainte temporelle relative au protocole de communication intéresse l'opérande gauche $(\varphi \text{ and } \psi)$ qui selon PSL_{ss} doit être booléen. La contrainte temporelle spécifie que ψ doit se produire X cycles après la satisfaction de φ . D'après L , toutes les deux doivent se produire avant ζ . Le raisonnement suivi pour la transformation formelle vers le sous ensemble simple nous permet d'obtenir la règle de transformation T_2 citée dans 4.3.1.1.

$T_2 : (\varphi \text{ and } \psi) \mathbf{before!} \zeta \xrightarrow{C} (\mathbf{prev}(\varphi, X) \text{ and } \psi) \mathbf{before!} \zeta$ $C : \psi \text{ occurs } X \text{ cycles after } \varphi$
--

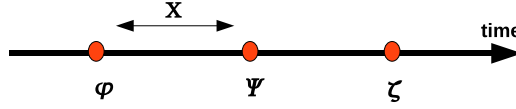
Transformation formelle vers PSL_{ss}

La formule correspondant au comportement attendu serait la SERE suivante :

$$v \models \{(\neg\zeta)[*] ; \neg\zeta \wedge \varphi ; (\neg\zeta \wedge \neg\psi)[*(X-1)] ; \neg\zeta \wedge \psi ; [*]\}$$

En effet, la contrainte temporelle à intégrer se traduit par le comportement temporel illustré par la Figure 4.9. Cette Figure décrit que φ doit se produire X cycles avant ψ et que cette dernière doit avoir lieu avant ζ .

Cette transformation peut être suffisante pour ramener l'expression L raffinée au sous ensemble simple de PSL . Voyons ci-dessous comment il est toutefois possible de la trans-

FIGURE 4.9 – Comportement temporel décrit par la contrainte temporelle de T_2

former, dans le cas où on ne souhaite pas utiliser de SERE (c'est le cas dans la version originale d'ISIS) :

$$v \models \{(\neg\zeta)[*] ; \neg\zeta \wedge \varphi ; (\neg\zeta \wedge \neg\psi)[*(X-1)] ; \neg\zeta \wedge \psi ; [*]\}$$

i.e., $\exists k < |v| - X$ s.t. $v^{k..} \models (\neg\zeta \wedge \varphi)$ and $v^{k+X} \models (\neg\zeta \wedge \psi)$ and $\forall j \leq k + X, v^j \models \neg\zeta$

D'après les formules précédentes, nous avons d'un côté :

$$\forall j \leq k + X, v^j \models \neg\zeta \text{ et } v^{k..} \models \neg\zeta \text{ et } v^{k+X} \models \neg\zeta.$$

Nous pourrions alors simplifier ces expressions en les remplaçant par la formule :

$$\forall j \leq k + X, v^j \models \neg\zeta.$$

Nous obtenons au final la formule suivante :

$$\text{i.e., } \exists k < |v| - X \text{ s.t. } v^k \models \varphi \text{ and } v^{k+X} \models \psi \text{ and;} \\ \forall j \leq k + X, v^j \models \neg\zeta$$

Posons $k' = k + X$. La formule devient :

$$\text{i.e., } \exists k' < |v| \text{ s.t. } v^{k'-X} \models \varphi \text{ and } v^{k'} \models \psi \text{ and} \\ \forall j \leq k', v^j \models \neg\zeta$$

La formule " $v^{k'-X} \models \varphi$ " nous ramène à la définition de l'une des fonctions de la couche booléenne de PSL³ dite "**prev**(a,num,c)". Cette fonction retourne la valeur d'une expression a num cycles auparavant, sur une trace échantillonnée par l'horloge c . Par défaut, la trace est échantillonnée selon le contexte de la propriété PSL dans laquelle cette expression apparaît. En s'appuyant sur cette définition, la formule peut être exprimée comme suit :

$$\text{i.e., } \exists k' < |v| \text{ s.t. } v^{k'} \models (\text{prev}(\varphi, X) \wedge \psi \wedge \neg\zeta) \text{ and} \\ \forall j < k', v^j \models \neg\zeta$$

$$\text{i.e., } v \models (\text{prev}(\varphi, X) \text{ and } \psi) \text{ before! } \zeta$$

³Dans PSL ces fonctions sont appelées "*Built-in functions*" et permettent d'exprimer des conditions particulières sur l'expression donnée en argument.

Pour les autres règles, les raisonnements suivis étant semblables à ceux présentés ici pour T_1 et T_2 , les preuves de transformation seront placées en annexe.

Transformation T_3

Nous considérons le raffinement de l'expression " $L : \varphi$ **before!** ψ " sous la contrainte temporelle C indiquant que ψ **doit avoir lieu X cycles après le début de la trace sur laquelle la formule φ **before!** ψ est évaluée**, (avec $X > 1$).

Contrairement aux deux cas précédents, cette contrainte sert à préciser qu'une action donnée est attendue dans un certain nombre de cycles. Dans ce cas, la formule "**next!** $[X]\psi$ " correspond à la description de la contrainte sous la forme d'une formule temporelle. L'intégration de cette contrainte dans le modèle de formule L se fait à l'aide de l'opérateur de conjonction logique. Néanmoins, la formule " φ **before!** ψ and **next!** $[X]\psi$ " résultante peut être rendue plus précise.

En passant par le remplacement des opérateurs **FL before!** et **next!** $[X]$ par leurs définitions formelles, on obtient :

$$\mathbf{next_e!}[0..X-1](\varphi) \text{ and } \mathbf{next_a!}[0..X-1](\neg\psi) \text{ and } \mathbf{next!}[X](\psi) .$$

Le détail de la transformation de cette formule est décrit dans l'annexe (*c.f.* annexe A.1). Au final la règle de transformation est (4.3.1.1) :

$T_3 : \varphi \text{ before! } \psi \xrightarrow{C}$ $\mathbf{next_e!}[0..X-1](\varphi) \text{ and } \mathbf{next_a!}[0..X-1](\neg\psi) \text{ and } \mathbf{next!}[X](\psi)$ $C : \psi \text{ occurs in } X \text{ cycles}$

4.3.1.2 Règles de transformation impliquant l'opérateur **until**

Nous nous intéressons dans cette section à la transformation des formules de la forme de $L : \varphi$ **until!** ψ en utilisant deux contraintes temporelles. Ces contraintes permettent de donner une précision temporelle sur le délai dans lequel ψ doit se produire.

Transformation T_4

La première contrainte " $C : \mathbf{next!}(\psi)$ " indique que l'opérande ψ se produit exactement un cycle après le début de la trace sur laquelle la formule φ **until!** ψ est évaluée. Si on reprend l'exemple de la propriété **always** (*req until! ack*) énoncée dans le chapitre 3 (*c.f.* Figure 3.5), la contrainte pourra indiquer dans ce cas que l'acquiescement est en fait attendu un cycle après l'envoi de la requête *req*. L'insertion de la contrainte temporelle permet la transformation de la formule L selon la règle donnée ci-dessous (équation 4.3.1.2) :

$$\begin{array}{l}
T_4 : \varphi \text{ **until!** } \psi \xrightarrow{C} \varphi \text{ and **next!** }(\psi) \\
C : \psi \text{ occurs 1 cycle after } \varphi
\end{array}$$

La preuve de transformation de la règle L est disponible en annexe (*c.f.* annexe A.2).

Transformation T_5

Dans ce cas, la contrainte est “ $C : \text{**next!**}[X](\psi)$ ”, $X > 1$. La preuve formelle de transformation de cette règle suite à l’inclusion de la contrainte temporelle est disponible en annexe (*c.f.* annexe A.3). La règle de transformation résultante est donnée ci-dessous (équation 4.3.1.2) :

$$\begin{array}{l}
T_5 : \varphi \text{ **until!** } \psi \xrightarrow{C} \text{**next.a!**}[0..X-1](\phi) \text{ and **next!**}[X](\psi) \\
C : \psi \text{ occurs } X \text{ cycles after } \varphi
\end{array}$$

4.3.1.3 Règles de transformation impliquant l’opérateur d’implication logique

Dans cette partie, nous nous intéressons à la transformation des formules PSL de la forme $L : (\varphi \text{ and } \psi) \rightarrow \zeta$ sous la contrainte temporelle indiquant que ψ se produit X cycles d’horloges après la satisfaction de φ . La partie gauche de la formule temporelle L deviendrait alors $\varphi \text{ and **next!**}[X]\psi$. La preuve de la règle de transformation est donnée en annexe A.4. Le résultat de la transformation est décrit par l’équation suivante (*c.f.* programme 1) :

$$\begin{array}{l}
T_6 : (\varphi \text{ and } \psi) \rightarrow \zeta \xrightarrow{C} \varphi \rightarrow \text{**next!**}[X](\psi \rightarrow \zeta) \\
C : \psi \text{ occurs } X \text{ cycles after } \varphi
\end{array}$$

Nous proposons de reprendre l’exemple de la propriété P_9 du `pv_dma` (*c.f.* programme 5.1) afin de montrer un aperçu de l’application des règles de transformation sur une assertion PSL. Intéressons nous d’abord à l’opérande gauche de l’opérateur d’implication logique.

La contrainte peut indiquer que la donnée est envoyée un cycle après l’envoi de l’adresse conformément au protocole AHB c’est à dire que :

“`cpu_initport.write.p2 == START`” est évaluée 1 cycle après “`cpu_initport.write_CALL()`
`&& cpu_initport.write.p1 == CTRL_REG.`”

La variable ζ de la règle de transformation T_6 correspond à la formule temporelle constituant l’opérande droit de l’opérateur d’implication logique.

L’application de la règle de transformation T_6 sur la propriété P_9 produit la propriété 4.2.

```

vunit P9_transT6 {
  // HDL_DECLs :
  const int CTRL_REG = 12+16384; //the DMA control register
  const int START = 1; // the START value
  // HDL_STMTs :

  // PROPERTY :
  assert
  always ((cpu_initport.write_CALL() &&
           cpu_initport.write.p1 == CTRL_REG
           -> next! (cpu_initport.write.p2 == START))
          -> next (dma_irq
                  before! (cpu_initport.write_CALL() &&
                           cpu_initport.write.p1 == CTRL_REG &&
                           cpu_initport.write.p2 == START))
          );
}

```

PROGRAMME 4.2 – Application de T6 sur l’assertion P9

Nous constatons que l’assertion résultante est partiellement transformée. En effet, la même contrainte temporelle doit aussi être prise en compte pour l’opérateur **before!**. L’application de la règle T_1 donne la forme finale de la transformation temporelle de l’assertion (c.f. programme 4.3) :

```

vunit P9_transT6T1 {
  // HDL_DECLs :
  const int CTRL_REG = 12+16384; //the DMA control register
  const int START = 1; // the START value

  // HDL_STMTs :

  // PROPERTY :
  assert
  always
    ( (cpu_initport.write_CALL() &&
      cpu_initport.write.p1 == CTRL_REG
      -> next! (cpu_initport.write.p2 == START))
      -> next (
        ((cpu_initport.write_CALL() &&
          cpu_initport.write.p1 == CTRL_REG )
          -> next! (!(cpu_initport.write.p2 == START)))
        until! (dma_irq)
        &&
        next_event! (dma_irq)(
          (cpu_initport.write_CALL() &&

```

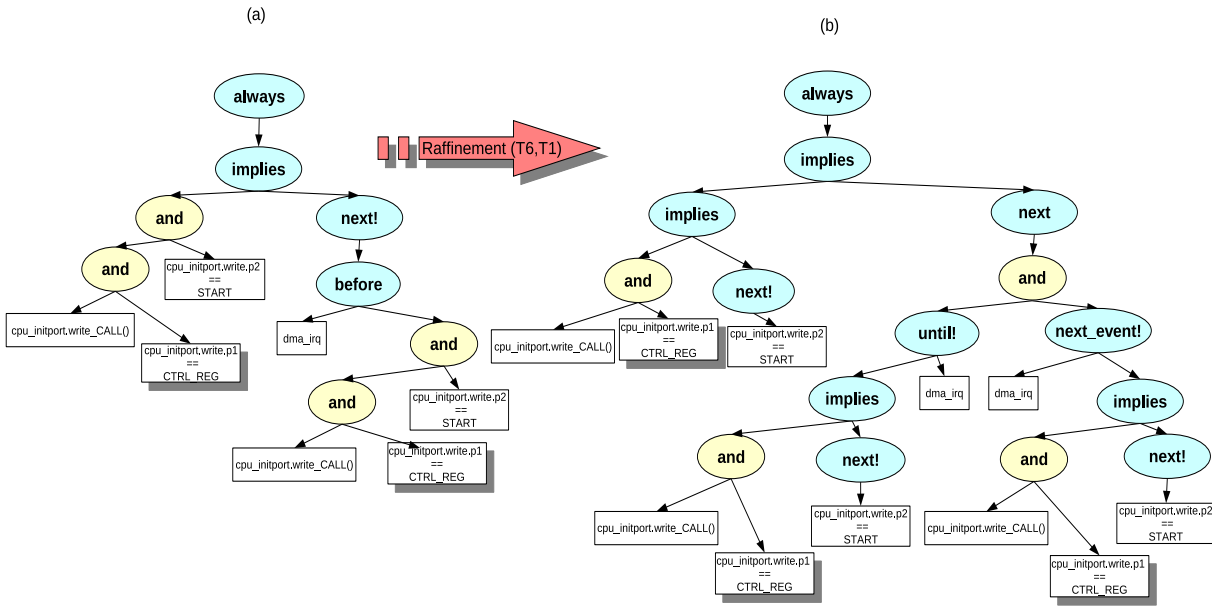
```

        cpu_initport.write.p1 == CTRL_REG )
        -> next! ( !
            (cpu_initport.write.p2 == START)))
    );
}

```

PROGRAMME 4.3 – Application de T1 sur l’assertion P9_transT6

La transformation temporelle des assertions peut impliquer une complexification structurelle non négligeable des assertions comme l’illustre la représentation des arbres syntaxiques de la Figure 4.10, d’où l’intérêt de l’automatisation de cette transformation. Dans le cas de l’assertion $P9$, par exemple, le nombre des opérateurs temporels passe de 4 à 11.

FIGURE 4.10 – La complexification structurelle due au raffinement temporel de l’assertion $P9$

4.3.2 Alternative : utilisation des SEREs

Nous avons présenté un premier ensemble de règles de transformation permettant la prise en considération des détails temporels concernant certains types de communication. Dans certains cas, l’obtention de la formule raffinée passe par une description séquentielle intermédiaire présentée sous la forme d’une SERE avant d’arriver à une formule temporelle transformée appartenant à PSL_{ss} . C’est le cas actuellement pour les règles T_2 et T_6 .

Il peut également être envisagé de conserver ces formules sous forme de SEREs. Au niveau RTL, la génération des moniteurs à partir des SEREs peut être faite en utilisant l’outil Horus. Une extension des capacités d’ISIS pour proposer un support pour les

SERES sera présentée dans la section 5.3. ISIS pourra ainsi être simplement utilisé pour les systèmes hybrides TLM/RTL.

4.4 Bilan

La réutilisation du code de vérification à différents niveaux d'abstraction est un facteur important dans la réussite du processus de vérification. La réutilisation de ces assertions pour la vérification des modèles moins abstraits est nécessaire pour confirmer que les exigences et les contraintes définies pour la vérification du modèle de haut niveau ont été respectées et maintenues aux niveaux inférieurs jusqu'à l'implémentation finale. Dans ce chapitre, nous avons exposé une analyse des exigences et des contraintes liées au raffinement des assertions transactionnelles en assertions RTL. Une approche du raffinement temporel a été présentée suite à une étude des principales solutions proposées dans l'état de l'art.

L'automatisation du raffinement réduit le temps de réécriture des assertions et garantit la cohérence syntaxique et sémantique des assertions raffinées. Cependant, l'analyse des contraintes du raffinement a démontré que, de même que pour le processus de raffinement des modèles dans le flot de conception, le raffinement des assertions ne peut pas être entièrement automatisé. Des informations supplémentaires sont indispensables pour ce processus ce qui rend nécessaire l'intervention du concepteur. En revanche l'intégration de ces informations dans l'assertion en vue de générer l'assertion raffinée doit être entièrement automatisée.

Dans le chapitre suivant, nous proposons de présenter la mise en œuvre du processus de transformation proposé ainsi que l'implémentation de l'extension de l'outil ISIS permettant la prise en charge des propriétés hybrides et des SERES.

Mise en oeuvre

Sommaire

Introduction	136
6.1 Cas d'étude : Plateforme de décodage vidéo Motion—JPEG	136
6.1.1 Plateforme et assertions transactionnelles	136
6.1.1.1 P_1 : Intégrité des données	137
6.1.1.2 P_2 : Non perte des données	139
6.1.2 Plateforme et assertions RTL	140
6.1.2.1 Norme Wishbone	140
6.1.2.2 Raffinement des assertions	141
6.2 Cas d'étude : Plateforme de compression d'images	144
6.2.1 Plateforme et assertions transactionnelles	144
6.2.1.1 P_3 : Configuration du DMA_a	145
6.2.1.2 P_4 : Configuration de la FFT	147
6.2.1.3 P_5 : Non perte des paquets en entrée	148
6.2.2 Plateforme et assertions raffinées	149
6.2.2.1 La norme AMBA-AHB	149
6.2.2.2 Le raffinement des assertions	150
6.2.2.3 Exemple de raffinement en utilisant les SEREs	159
6.3 Résultats et analyses	161
6.3.1 Complexification des assertions raffinées	161
6.3.2 Analyse des performances	162
6.3.2.1 Plateforme de décodage vidéo (MJPEG)	163
6.3.2.2 Plateforme de compression d'images	164
6.3.3 Synthèse	169
6.4 Bilan	169

“Les détails font la perfection, et la perfection n’est pas un détail.”

Léonard de Vinci

Introduction

Ce chapitre est dédié à la présentation de l’outil prototype de raffinement des propriétés et à l’implémentation des SERES dans ISIS. Nous allons y présenter l’implémentation des principes et des propositions énoncés lors du chapitre précédent. Le contenu de ce chapitre a fait l’objet de ces deux publications : [BHAPB14a] et [BHAPB14b].

La Figure 5.1 est une vue abstraite de la méthodologie de vérification proposée. Cette figure permet de localiser l’outil de raffinement (*Refinement Tool*) dans l’infrastructure de vérification.

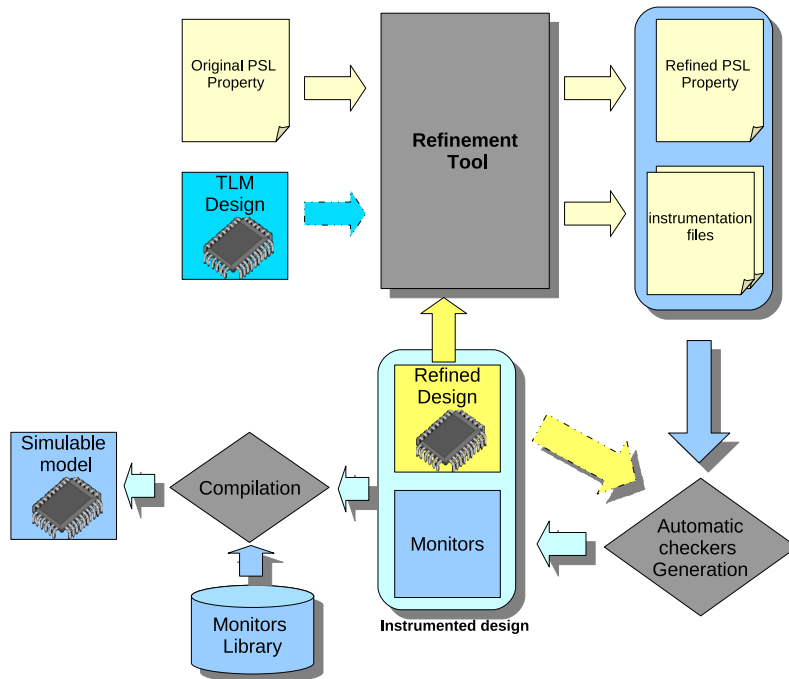


FIGURE 5.1 – Méthodologie de vérification proposée

Globalement, l’outil prend en entrée un ensemble de propriétés PSL ainsi que les modèles de la plateforme décrits au niveau TLM et RTL et produit en sortie les propriétés PSL raffinées ainsi que les fichiers d’instrumentation associés. Les fichiers d’instrumentation font le lien entre les assertions produites et le modèle à surveiller. Concrètement, ils établissent les correspondances entre les identifiants utilisés dans les propriétés et les composants réellement présents dans le modèle à surveiller. Les modèles de la plateforme ne font pas partie intrinsèque du processus de raffinement mais ils servent à la validation de certaines données fournies par l’utilisateur.

Par la suite, il est possible de faire appel à ISIS ou Horus pour la génération automatique des moniteurs de surveillance à partir des assertions PSL raffinées. L'association des moniteurs au modèle à surveiller forme le modèle instrumenté. La compilation de ce modèle en utilisant la bibliothèque des moniteurs élémentaires donnera le modèle à simuler.

L'outil de raffinement des assertions a été développé durant le travail de thèse. Des modifications ont aussi été apportées sur la bibliothèque des moniteurs et sur l'outil ISIS afin de fournir le support nécessaire pour les assertions hybrides (TLM-RTL) et les SEREs.

Le détail des interventions sur ces trois parties sera donné au cours de ce chapitre.

5.1 Outil de raffinement des propriétés PSL

L'outil prototype est un programme développé en langage Java¹ et intégré dans ISIS afin de conserver le même environnement durant la plus grande partie du flot de vérification. Cette union a aussi d'autres avantages puisqu'elle permet de donner l'accès à certaines fonctionnalités déjà présentes dans ISIS principalement liées à l'analyse syntaxique des propriétés PSL et des modèles. Cependant, il est possible d'extraire l'outil de raffinement de cet environnement. Comme le montre la Figure 5.2, l'outil met en œuvre une base de données de règles de transformation (*c.f.* section 4.3.1) et un ensemble d'analyseurs syntaxiques (*parser*). Le format de stockage des règles de transformation est un fichier XML. Hormis le fait qu'il est standard, ce format est particulièrement adapté pour le stockage des données arborescentes et structurées. Le modèle de ce fichier est fixé en utilisant un langage de description standard appelé XSD² permettant de définir la structure et le type du contenu du document XML. Ce fichier est représenté en annexe (*c.f.* annexe B.1).

Selon ce modèle, chaque règle de réécriture (identifiée par la balise *rule*) est composée d'un modèle de formule "source" notée *source* correspondant à la partie gauche (*L*) de la règle de transformation, d'un modèle de formule "destination" étiquetée *dest* correspondant à la partie droite (*R*) de la règle de transformation, et d'une contrainte temporelle (*C*) notée "*constraint*".

Toutes les opérations extérieures au processus de transformation (délimité par la boîte "*Transformation Process*" dans la Figure 5.2) sont considérées comme étant des étapes de pré-raffinement.

Durant le flot de raffinement, l'utilisateur est amené à fournir certaines informations et à faire des choix. Afin de faciliter ces tâches et les rendre plus ergonomiques, notre outil fait appel à des interfaces graphiques ainsi qu'à des fichiers Excel pré-remplis.

¹27000 lignes de code en Java

²XML *Schema Description* est un langage de description de format de document XML. <http://www.w3.org/standards/xml/schema>

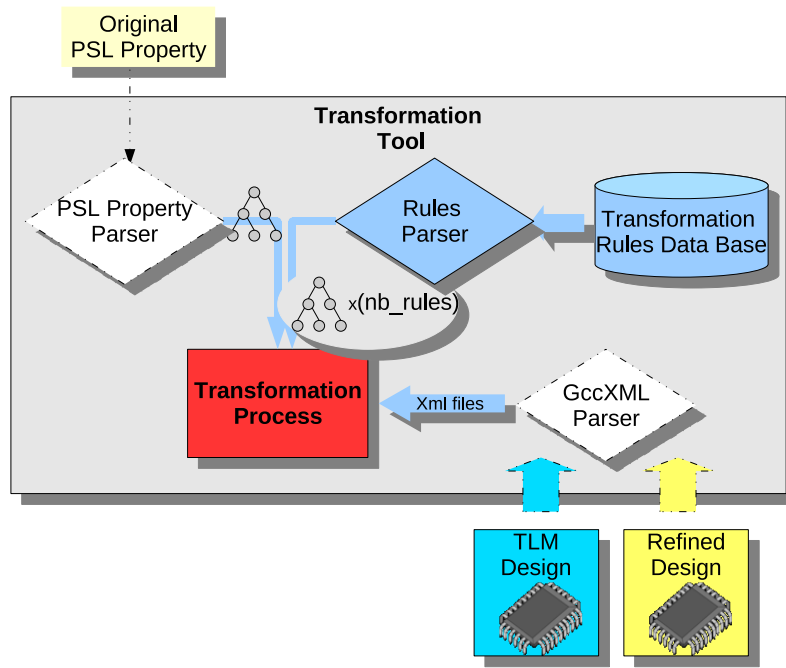


FIGURE 5.2 – La composition de l'outil

Dans les parties qui suivront nous allons nous focaliser sur les principales étapes du flot de raffinement. Nous commencerons par une description succincte des étapes de pré-raffinement avant de passer à celle du processus de raffinement. Nous nous appuierons dans nos explications sur l'exemple de l'assertion *P9* de la plateforme *pv_dma* que nous avons spécifiée dans le Programme 5.1.

```

vunit P9 {
    // HDL_DECLS :
    const int CTRL_REG = 12+16384; //the DMA control register
    const int START = 1; // the START value

    // HDL_STMTs :

    // PROPERTY :
    assert
    (always ( (cpu_initport.write_CALL() &&
              cpu_initport.write.p1 == CTRL_REG &&
              cpu_initport.write.p2 == START)
            ->next(dma_irq
                  before! (cpu_initport.write_CALL() &&
                          cpu_initport.write.p1 == CTRL_REG &&
                          cpu_initport.write.p2 == START)))
    )@(dma_irq.write_CALL() || cpu_initport.write_CALL());
}

```

PROGRAMME 5.1 – La propriété P9 pour le *pv_dma*

5.1.1 Les étapes de pré-raffinement

Nous avons distingué dans la Figure 5.2 trois analyseurs syntaxiques :

1. l'analyseur syntaxique GCC-XML³ destiné aux modèles de plateformes codés en SystemC (C++),
2. l'analyseur syntaxique des propriétés PSL transactionnelles,
3. l'analyseur syntaxique des règles de transformation.

Les premières opérations consistent à analyser les architectures des modèles donnés en entrée de l'outil de transformation dans le but de produire une représentation structurée et hiérarchique des composants et de leurs méthodes. Cette représentation hiérarchique utilise encore une fois XML comme format de stockage. La fonction effectuant ce traitement fait partie d'ISIS. Elle fait appel à l'analyseur du code C++ appelé GCC-XML pour la génération d'une description XML intermédiaire non hiérarchique.

L'outil utilise aussi l'analyseur syntaxique des propriétés PSL développé dans ISIS afin de générer l'arbre syntaxique de la propriété PSL transactionnelle[Fer11].

Nous avons développé l'analyseur syntaxique dédié aux règles de transformation afin de générer les arbres syntaxiques associés. Il faut néanmoins noter que la base de données des règles de transformation ne sera pas nécessairement analysée pour chaque transformation. Les arbres syntaxiques des règles sont sauvegardés sous format binaire selon une technique dite "sérialisation"[Dou99]. La mise à jour du flux binaire se fait automatiquement suite à la mise à jour de la base de données des règles de transformation. L'opération inverse est dite "dé-sérialisation" et elle permet de reconstituer les objets à partir du flux binaire.

5.1.2 Le processus de raffinement

Analyse des propriétés transactionnelles

Le processus de raffinement commence par l'identification automatique de certaines informations :

- les identifiants des composants impliqués dans les communication mises en jeu dans l'assertion,
- les fonctions de communication observées dans l'assertion et leur paramètres,
- les points d'observation des fonctions ("CALL", "END").

Dans le cas de la propriété *P9*, ceci revient à extraire la fonction `write` et ses deux premiers paramètres (`p1` et `p2`) ainsi que le point d'observation "CALL" pour le composant CPU ; et la fonction `write` du signal `dma_irq`, son paramètre `p1` et son point d'observation "CALL".

³C'est une extension du compilateur GCC qui permet la génération d'une description XML à partir d'un programme C++ en utilisant la représentation interne de GCC. <http://gccxml.github.io/HTML/Index.html>

Choix des fonctions de communication et identification structurale

En se basant sur ces informations, l'outil demande à l'utilisateur d'indiquer les fonctions modifiées dans le design raffiné via l'interface graphique décrite dans la Figure 5.3. Dans le cas général, cette interface liste toutes les fonctions observées dans la propriété. Dans l'exemple de la propriété *P9*, il s'agit en particulier de la fonction `write` du port *initiator* du CPU et de la fonction `write` du signal `dma_irq`. Comme le montre la Figure 5.3, nous supposons que le raffinement de la plateforme a uniquement affecté l'interface du CPU.

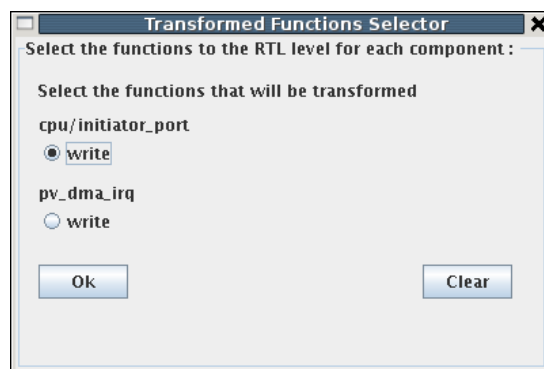


FIGURE 5.3 – Interface de sélection

En fonction de ces choix, un ensemble de fichiers Excel pré-remplis va être automatiquement généré. Chaque fichier correspond à un composant observé de la(es) propriété(s) à transformer. La structure du fichier diffère selon la nature de la transformation du composant (pas de transformation, transformation). La Figure 5.4 donne un exemple.

Dans la figure 5.4, toutes les données en dehors des zones entourées en gras correspondent à des informations automatiquement générées par l'outil. Les champs pré-remplis sont souvent protégés en écriture afin de prévenir les incohérences des données.

Selon cette figure, le fichier Excel (a) correspondant au signal `pv_dma_irq` (associé à la variable “`dma_irq`” dans l’assertion) vise uniquement à demander les informations permettant d’identifier le composant dans la plateforme raffinée (nom et type). L’utilisateur pourra aussi, optionnellement, spécifier un nouvel identifiant pour ce composant dans l’assertion raffinée.

Dans le second cas, l’outil génère un fichier Excel contenant toutes les fonctions raffinées observées dans les différentes propriétés transactionnelles. Pour chaque fonction, l’outil précise les points d’observation (“CALL”, “END”), ainsi que les paramètres utilisés dans ces propriétés. L’outil demande à l’utilisateur de spécifier l’ensemble des signaux impliqués dans chaque opération selon le protocole qu’il a implémenté dans la plateforme raffinée. Ces signaux reflètent, dans la plupart des cas, l’une des interfaces d’un protocole de communication en particulier.

Dans l’exemple décrit par la figure 5.4 (b), l’utilisateur indique qu’un appel à la fonc-

tion d'écriture transactionnelle : `write()` du port *initiator* du CPU c'est à dire début d'une opération d'écriture simple par le CPU, met en jeu au niveau RTL les signaux : `cpu_HGRANT`, `cpu_HWRITE`, `cpu_HTRANS`, `cpu_HBURST`, `cpu_HSIZE`, et `router_HREADY`.

Ce sont les signaux que le concepteur de la plateforme raffinée estime être significatifs pour caractériser ce début d'opération d'écriture. Dans le cas de cet exemple, le protocole suivi est le protocole AMBA-AHB. Selon ce protocole, le début d'une opération d'écriture simple, telle celle décrite ici, nécessite d'avoir, au préalable, l'autorisation de l'arbitre du bus (`cpu_HGRANT`) ainsi que la confirmation de la disponibilité de la cible (`router_HREADY`).

Le début effectif de l'opération a lieu quand le signal `cpu_HTRANS` indique le début de la phase d'adresse et que le signal `HWRITE` indique une opération d'écriture. Le signal `HBURST` indique la présence ou l'absence de rafale et `HSIZE` donne la taille des données transférées sur le bus de données. Une description plus détaillée sur ce protocole sera donnée dans le prochain chapitre (*c.f.* section 6.2.2.1).

De la même manière, l'utilisateur fournit les correspondances structurelles entre les paramètres de chaque fonction et les signaux de la plateforme.

L'outil demandera en plus, la spécification de l'horloge et du front⁴ de synchronisation pour les communications.

Il faut souligner que seules les informations dont l'obtention n'est pas automatisable sont demandées à l'utilisateur. L'outil a été conçu de manière à faciliter cette tâche et à réduire au maximum le risque d'erreurs.

La vérification des informations introduites consiste à s'assurer de la présence des composants et des signaux mentionnés dans la plateforme raffinée.

Transformation des variables globales

Cette étape permet de mettre à jour les variables globales dans la couche de modélisation et aussi dans l'assertion. Trois types de modifications sont proposées : la suppression de la variable, son remplacement par une autre expression ou le changement du type et de la valeur de la variable. L'utilisateur a accès à ces opérations via un tableau Excel tel que celui décrit par la Figure 5.5.

L'outil commence par collectionner l'ensemble des variables globales de l'assertion⁵. Les constantes sont exclues de ce traitement. Ensuite, si l'ensemble n'est pas vide, un fichier Excel pré-rempli semblable à celui de la figure 5.5 sera généré pour chaque propriété à raffiner. Chaque variable déclarée dans la propriété transactionnelle est placée dans une case de la colonne "*Variable Name*". À chaque variable correspondent trois cases :

⁴Cette information est optionnelle. Par défaut, le front sélectionné est le front montant (*posedge*).

⁵L'outil ne fournit pas un analyseur (*parser*) pour la couche de modélisation de l'assertion.

Component equivalence at RTL level			
2014/07/01 17:00:29			
	Variable Name	Component Name	Component type
TLM	dma_irq	pv_dma_irq	sc_signal<bool>
RTL		pv_dma_irq	sc_signal<bool>
NB: if no changes then don't fill cells.			

(a)- « pv_dma_irq » Excel file -

Component equivalence in RTL design for TLM			
2014/07/01 17:00:29			
	Variable Name	Component Name	Component type
TLM	cpu_initiator_port	cpu/initiator_port	pv_initiator_port
RTL		cpu	dma_testbench
NB: if no changes don't fill cells.			
Give concerned signals for each method			
CLOCK	clk		
Edge(posedge/negedge)			
Input	Type	RTL signals	
write_CALL0	public	cpu_HGRANT; cpu_HWRITE; cpu_HTRANS; cpu_HBURST; cpu_HSIZE; router_HREADY	
write.p1	int	cpu_HADDR	
write.p2	int	cpu_HWDATA	

FIGURE 5.4 – Exemple de fichier Excel généré

- La colonne “*Replace in RTL with Input*” correspond à l’opération de remplacement de la variable par l’expression introduite par l’utilisateur. La variable est alors éliminée de la partie déclarative de la couche modélisation et elle est remplacée, dans la couche modélisation et dans le corps de l’assertion, par la nouvelle expression.
- La colonne “*RTL variable Type*” permet le changement du type de la variable dans la propriété raffinée. Ce changement affecte uniquement la partie déclarative de la couche de modélisation.
- La colonne labellisée “*RTL variable value*” permet d’affecter une expression à la variable. Cette opération est souvent associée au changement de type de la variable. Dans ce cas, la modification ne touche qu’à la couche de modélisation et intéresse principalement les opérations d’affectation de cette variable.

Si aucune des trois cases correspondant à ces colonnes n’est remplie pour une variable donnée, alors l’outil déduit que cette variable doit être supprimée dans la propriété raffinée. Afin d’exécuter cette opération, l’outil exige que la variable ne soit pas utilisée dans le corps de l’assertion.

Dans l’exemple de la propriété *P9*, aucune variable globale n’est utilisée. Cette étape est par conséquent ignorée. Le chapitre 6 contient un exemple d’utilisation de ce fichier (*c.f.*

Property Variables Transformation			
* If the variable is not used at the RTL level then don't fill any field.			
* If the variable is replaced by an other variable or is equivalent to another input then give this variable/input into the second field and don't fill the variable type field			
* If the variable is used at the rtl level then give it's new type and it's value			
Global Variables of the Property :	prop1		
Variable Name	replace in RTL with input	RTL Variable Type	RTL Variable Value
data_ptr			
value			

FIGURE 5.5 – Exemple de fichier Excel de transformation des variables

page 153).

Saisie des chronogrammes

Suite à la vérification des informations introduites par l'utilisateur, l'outil procède à la saisie des chronogrammes des signaux spécifiés pour chaque fonction raffinée. Chaque chronogramme fait l'objet d'une nouvelle feuille de calcul semblable à celle décrite dans la Figure 5.6.

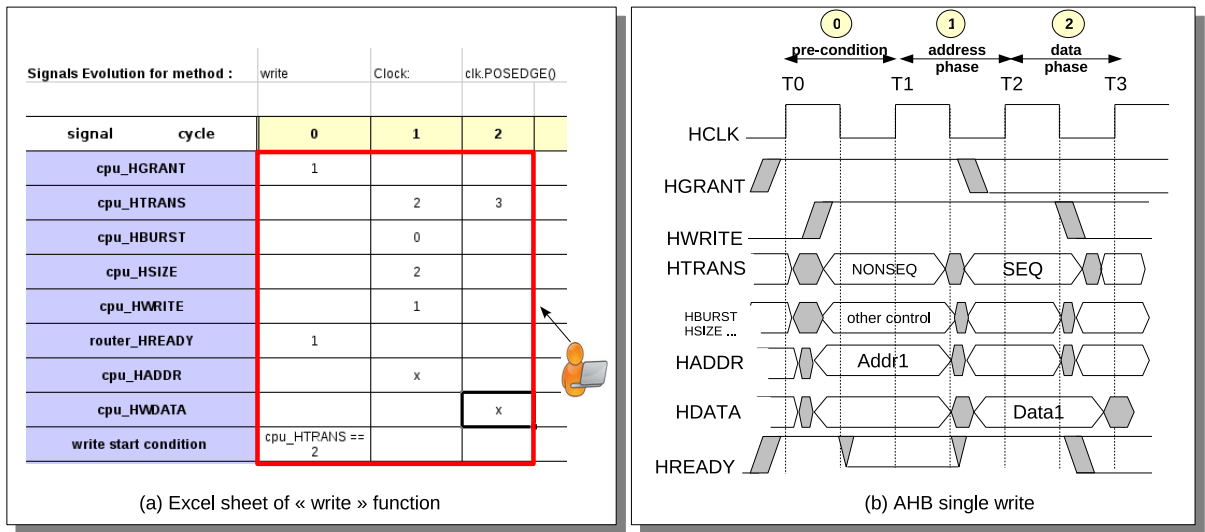


FIGURE 5.6 – Exemple d'un chronogramme Excel

L'outil génère les lignes correspondant aux signaux (et variables) pour l'opération observée ainsi que la structure du chronogramme. À cette étape, l'utilisateur doit fournir les informations sur l'évolution des signaux à chaque cycle de l'horloge spécifiée.

L'identification du cycle correspondant à un point d'observation de la fonction de communication ("CALL", "END") se fait en désignant, dans la dernière ligne du chronogramme ("write start condition"), n'importe lequel des signaux prenant une valeur significative dans ce cycle.

Pour notre exemple, le chronogramme correspondant à une opération d'écriture simple, tel que défini dans la norme est décrit dans la Figure 5.6(b). L'utilisateur retranscrit

simplement dans le fichier Excel les informations données par ce chronogramme.

Il faut souligner qu'à l'exception des signaux de contrôle, tous les autres signaux (adresse, données) doivent prendre une valeur non booléenne, simplement notée "x" par exemple dans la figure 5.6(a). Ceci indique que le signal correspondant est valide à ce cycle mais qu'il peut être associé à une valeur quelconque.

Selon le chronogramme 5.6(b), le cycle contenant le début effectif de l'opération d'écriture correspond à la phase d'adresse du protocole AHB. Cette phase correspond au cycle 1 dans le tableau 5.6(a). À cette phase, les valeurs des signaux valides sont :

`cpu_HWRITE = 1`, `cpu_HTRANS = 2` (ou `NONSEQ`), et `cpu_HBURST = 0`.

L'utilisateur a le choix entre ces expressions afin d'identifier le début d'une opération d'écriture simple dans le champ "**write start condition**". L'interprétation de ce chronogramme par l'outil permettra de déduire que le cycle correspondant à l'appel de la fonction `write` (`write_CALL`) est le cycle #1. Ce point temporel servira de repère lors de l'identification de la contrainte temporelle. La spécification d'un signal et non pas de la valeur du cycle est un choix conceptuel.

Le démarrage d'une opération de transfert nécessite au préalable que le maître soit autorisé par le bus et que l'esclave soit prêt à recevoir la requête (i.e. `cpu_HGRANT` et `router_HREADY` doivent être à l'état haut). Dans le chronogramme de l'opération d'écriture simple, ces signaux sont spécifiés au cycle #0, un cycle avant le début effectif de l'opération d'écriture (nous verrons plus loin que cela entraînera, dans l'assertion, l'utilisation de la fonction PSL "`prev`" permettant l'accès à des valeurs précédentes de signaux).

Génération des contraintes temporelles

La spécification du comportement des signaux permet à l'outil de déduire les contraintes temporelles relatives au protocole de communication implémenté au niveau RTL. La génération de la contrainte temporelle associée à une fonction de communication s'appuie principalement sur l'identification du début ou de la fin d'une opération. Les contraintes relatives aux fonctions de communication observées sont stockées dans un fichier XML tel que le décrit le programme 5.2. Ce fichier est spécifique à chaque propriété. Il regroupe l'ensemble des contraintes temporelles déduites à partir des chronogrammes des fonctions de communication raffinées dans cette propriété.

```

<property>P9</property>
<constraints>
  <A_X_delay_B>
    <A>cpu_initport.write_CALL() &&
      cpu_initport.write.p1 == VAR_p1</A>
    <B>cpu_initport.write.p2 == VAR_p2</B>
    <X>=1</X>
    <CLK>clk</CLK>
  </A_X_delay_B>
</constraints>

```

PROGRAMME 5.2 – Contrainte générée pour *P9*

Chaque fichier est composé d’une balise “property” contenant le nom de la propriété concernée et d’un ensemble de contraintes “<A_X_delay_B>” regroupées dans la balise “constraints”. Chaque contrainte est composée de deux expressions (A et B), d’un délai X et d’une horloge CLK. Elle signifie que A a lieu X cycles avant B selon le contexte de l’horloge CLK.

Le programme 5.2 contient la contrainte temporelle déduite à partir du chronogramme de la Figure 5.6 pour la conjonction de conditions mises en jeu dans l’assertion *P9*. Elle indique que l’envoi des données se fait un cycle après l’envoi de l’adresse et des signaux de contrôle. La contrainte est exprimée en fonction des expressions transactionnelles afin de pouvoir l’appliquer à la propriété transactionnelle. Les valeurs non définies du chronogramme (notées “x”) ont été remplacées par les variables VAR_p1 et VAR_p2.

Prise en compte des contraintes et unification

Comme le montre la Figure 5.7, l’application des règles de transformation passe par trois étapes : l’instantiation, l’unification et l’application des règles de transformation. Le résultat de ces étapes est une propriété PSL temporellement raffinée. Le traitement que nous allons décrire est itéré pour chaque contrainte de la propriété.

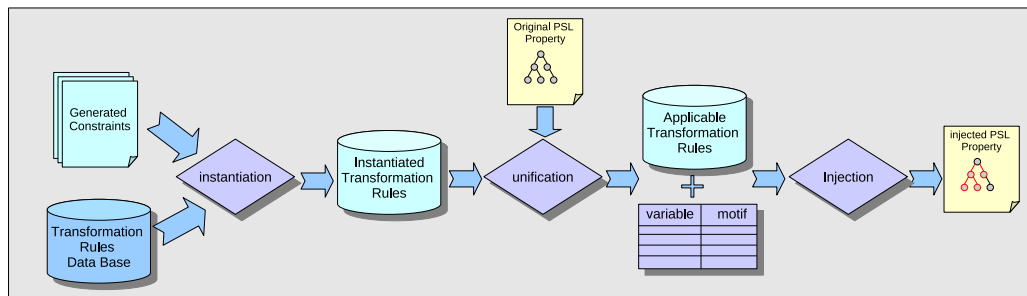


FIGURE 5.7 – Processus de la transformation temporelle

1. **L’instantiation** : cette opération consiste à intégrer la contrainte temporelle dans les règles de transformation. Concrètement, les “variables” (φ , ψ et ζ) de la con-

trainte C de chaque règle de transformation sont remplacées par les expressions booléennes de la contrainte temporelle générée. Ensuite, les expressions booléennes sont intégrées dans la partie gauche (L) et droite (R) de la règle de transformation. On obtient des règles de transformation partiellement instanciées. Considérons notre exemple du programme 5.2 et la règle de transformation (*c.f.* règle 1) :

$$T_6 : (\varphi \text{ and } \psi) \rightarrow \zeta \xrightarrow{C} \varphi \rightarrow \text{next}![X](\psi \rightarrow \zeta)$$

$$C : \psi \text{ occurs } X \text{ cycles after } \varphi$$

L'instantiation de la contrainte temporelle (C_{T_6}) de cette règle en utilisant la contrainte générée permet de déduire qu'il faut remplacer, dans T_6 , la variable φ par l'expression `(cpu_initport.write_CALL() && cpu_initport.write.p1 == VAR_p1)` et la variable ψ par l'expression `(cpu_initport.write.p2 == VAR_p2)`. Le résultat de l'instantiation de la partie gauche L_{T_6} et de la partie droite R_{T_6} de cette règle en utilisant la contrainte donnée dans le programme 5.2 est cité dans le programme 5.3 :

```

CT6 : (cpu_initport.write.p2 == VAR_p2) occurs 1 cycle after
(cpu_initport.write_CALL() && cpu_initport.write.p1 == VAR_p1)

LT6 : (cpu_initport.write_CALL() &&
        cpu_initport.write.p1 == VAR_p1 &&
        cpu_initport.write.p2 == VAR_p2)
        -> ζ

RT6 : ((cpu_initport.write_CALL() &&
        cpu_initport.write.p1 == VAR_p1
        -> next![1] ( cpu_initport.write.p2 == VAR_p2
        -> ζ)

```

PROGRAMME 5.3 – Exemple de règle de transformation instanciée

2. **Le filtrage (ou *pattern matching*)** : c'est une technique qui consiste à unifier une donnée avec une expression jouant le rôle de filtre et contenant éventuellement des variables. Son objectif est d'identifier les substitutions entre les **motifs** constituant la donnée et les **variables** du filtre de façon à les rendre symboliquement identiques. Chaque substitution est une association distincte $\{variable, motif\}$. Dans notre cas, la donnée est la propriété d'origine, les filtres sont les parties gauches (L) des règles de transformation partiellement instanciées issues de l'étape précédente.

Ayant une propriété PSL transactionnelle, l'algorithme de filtrage implémenté s'applique de manière itérative sur toute la base de données des règles de transformation. Pour chaque règle, la fonction de filtrage est appliquée récursivement sur l'arbre syntaxique de la propriété d'origine. Si une incompatibilité structurelle a été décelée, alors le filtrage échoue. Sinon, un ensemble de substitutions est retourné pour chaque

règle applicable sur la propriété d'origine.

L'unification de la règle de transformation partiellement instanciée citée dans le programme 5.3 avec la propriété $P9$, par exemple, donnera les substitutions suivantes : $\{\text{VAR_p1}, \text{CTRL_REG}\}$, $\{\text{VAR_p2}, \text{START}\}$ et $\{\zeta, \text{next}(\text{dma_irq} \text{ before! } \dots)\}$. L'ensemble des règles ayant des substitutions non nulles constituent l'ensemble des règles applicables sur la propriété à raffiner.

À la fin du processus de filtrage, il est possible d'avoir plusieurs règles applicables à l'intérieur d'une même propriété PSL. C'est le cas pour la propriété $P9$ et des deux règles T_1 et T_6 . La règle de transformation T_1 s'applique sur le nœud de l'arbre syntaxique de la propriété correspondant à l'opérateur “**before!**” ; tandis que la règle T_6 , s'applique sur le nœud correspondant à l'opérateur d'implication logique (*c.f.* figure 4.10(a)).

À la fin de cette étape, les substitutions obtenues sont appliquées aux parties gauches et droites des règles de transformation correspondantes afin d'obtenir des règles complètement instanciées.

3. **L'application des règles de transformation dans la propriété PSL** : cette étape consiste à remplacer, dans l'arbre syntaxique de la propriété d'origine, les formules qui correspondent à la partie gauche de la règle de transformation par la partie droite de cette règle. Un exemple est illustrée par la Figure 5.8. Cette figure décrit l'arbre syntaxique résultant de l'application de la règle de transformation T_6 dans la propriété $P9$. Nous rappelons que l'arbre syntaxique de la propriété d'origine a été donné dans la Figure 4.10(a).

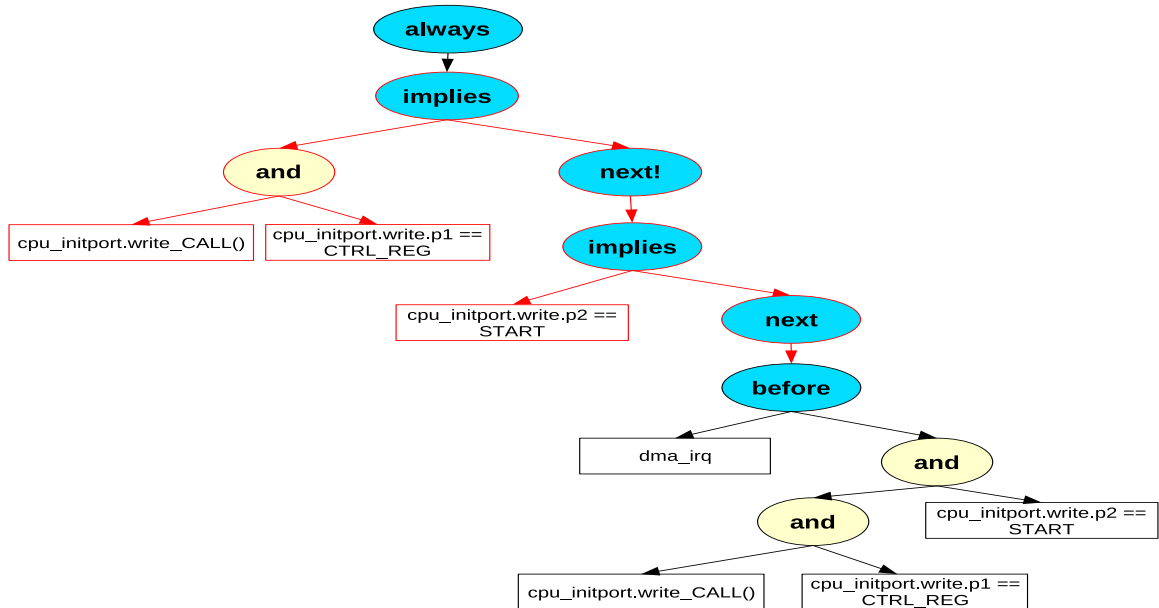


FIGURE 5.8 – Arbre syntaxique de l'application de T_6 dans $P9$

Dans le cas général, par définition, les règles sont injectées dans la propriété en

remontant dans l'arbre syntaxique. Ceci revient, dans notre exemple, à injecter la règle T_6 pour la transformation de l'opérateur d'implication logique et à appliquer ensuite la règle T_1 . Le résultat de l'application de ces deux règles dans P_9 a été décrit dans le programme 4.3 (c.f. page 103).

Transformation structurelle

La dernière étape du processus de transformation consiste à remplacer les expressions transactionnelles par les expressions portant sur les signaux équivalents et les valeurs associées. Cette correspondance est déduite à partir des données introduites par l'utilisateur dans les fichiers Excel fournis au début du processus de transformation.

Dans le cas de notre exemple, il s'agit de remplacer l'expression :

```
cpu_initport. write_CALL()
```

par l'expression de ses signaux correspondants :

```
prev (cpu_HGRANT ,1, clk) && prev (router_HREADY ,1, clk) && cpu_HWRITE  
&& cpu_HTRANS ==2 && cpu_HBURST ==0 && cpu_HSIZE ==2.
```

À la fin de cette étape, l'expression servant à l'échantillonnage de la trace de la propriété est mise à jour. L'horloge de synchronisation est introduite et les fonctions raffinées sont éliminées. Le résultat de cette étape constitue l'assertion en sortie du processus de raffinement. La propriété obtenue à partir de la propriété P_9 , est décrite dans le programme 5.4 :

```
vunit P9_refined {  
    // HDL_DECLs :  
    const int CTRL_REG = 12+16384; //the DMA control register  
    const int START = 1; // the START value  
  
    // HDL_STMTs :  
  
    // PROPERTY :  
    assert  
    (always  
        ( prev(cpu_HGRANT,1,clk) && prev(router_HREADY,1,clk) &&  
          cpu_HWRITE && cpu_HTRANS == 2 && cpu_HBURST == 0 &&  
          cpu_HSIZE == 2 && cpu_HADDR == CTRL_REG  
          -> next! (cpu_HWDATA == START  
              -> next(  
                  ((  
                      (prev(cpu_HGRANT,1,clk) && prev(router_HREADY,1,clk)  
                      &&  
                      cpu_HWRITE && cpu_HTRANS == 2 && cpu_HBURST == 0 &&  
                      cpu_HSIZE == 2 && cpu_HADDR == CTRL_REG )
```

```

        ->next! (! (cpu_HWDATA == START)))
            until! (dma_irq)
    )
    &&
    next_event!(dma_irq)
    (prev(cpu_HGRANT,1,clk) && prev(router_HREADY,1,clk)
    &&
    cpu_HWRITE && cpu_HTRANS == 2 && cpu_HBURST == 0 &&
    cpu_HSIZE == 2 && cpu_HADDR == CTRL_REG
    ->next! (! (cpu_HWDATA == START) )
    )
    )
    )
    )@ (clk.rising() || dma_irq.write_CALL() );
}

```

PROGRAMME 5.4 – Résultat du raffinement de P_9

Il faut noter que la fonction `rising()` est propre à la nouvelle version d'ISIS. Elle sert à exprimer que l'observation est faite sur les fronts montants de l'horloge `clk`. La trace d'évaluation du moniteur généré à partir de cette assertion sera échantillonnée sur les fronts montants de l'horloge `clk` et sur les appels de la fonction `write` du signal `dma_irq`. En effet, il s'agit d'un signal d'interruption asynchrone, ses notifications doivent générer des points dans la trace indépendamment de l'horloge.

Le raffinement structurel de la propriété augmente sa complexité syntaxique. L'assertion $P_9_refined$, par exemple, comporte 28 expressions booléennes contre 13 pour l'assertion $P_9_transT6T1$ résultant de la transformation temporelle (*c.f.* programme 4.3).

Une illustration complète et détaillée du flot de raffinement est fournie en annexe (*c.f.* annexe B.2).

5.1.3 Réutilisation des fichiers Excel

Il est possible de donner directement un ensemble de fichiers Excel ainsi que les fichiers XML contenant les contraintes temporelles associées en entrée du flot de transformation. Dans ce cas, après la vérification de leur contenu, le processus de raffinement démarre à partir de l'étape "Prise en compte des contraintes et unification".

L'outil permet aussi le raffinement incrémental des propriétés. Selon cette démarche, l'utilisateur peut fournir uniquement les fichiers Excel et XML associés aux composants précédemment raffinés. Sur cette base, l'outil générera de nouveaux fichiers Excel uniquement pour les nouveaux composants concernés par le raffinement ou permettra de compléter les fichiers existants. Dans le dernier cas, deux modifications sont possibles : la génération des feuilles de calcul pour les nouvelles fonctions raffinées, et la mise à jour

des feuilles de calcul des fonctions déjà raffinées (ajout de paramètres ou d'un point d'observation).

Ces possibilités permettent de réduire l'effort et le temps de transformation dans le flot de vérification.

5.2 Extensions d'ISIS

Dans cette section, nous nous positionnons en sortie de l'outil de raffinement (*c.f.* Figure 5.1). L'objectif est alors de générer les moniteurs de surveillance à partir des assertions raffinées (au niveau RTL ou hybrides).

5.2.1 Prise en compte des fronts montants ou descendants de signaux autres que des signaux de type horloge

Les conditions booléennes peuvent ne pas simplement porter sur le fait qu'un signal booléen soit actuellement à `true` ou à `false`, mais sur le fait que le signal avait une certaine valeur un certain nombre de cycles auparavant ou qu'il soit précisément en train de passer à `true` (front montant) ou à `false` (front descendant). C'est le cas, par exemple des signaux `cpu_HGRANT` et `router_HREADY` dans l'assertion 5.4. La validation de l'expression nécessite l'accès à la valeur de ces signaux, au cycle précédent.

Pour gérer ce cas, ISIS va automatiquement générer, pour les signaux pour lesquels ce type de conditions devra être utilisé, respectivement, les prédicats “prev”, “rose” et “fell” qui s'apparentent aux fonctions prédéfinies correspondantes dans PSL[PSL05]. Il sera donc possible d'écrire dans l'assertion, une condition booléenne de la forme `rose(sig)` ou `rose(sig,clk)` pour indiquer que la présence d'un front montant sur le signal booléen `sig` est requise. Nous verrons ce cas, au prochain chapitre, dans un exemple pour le protocole Wishbone section 6.1.2.2.

5.2.2 Possibilité de consulter les valeurs des signaux

Dans le contexte transactionnel, des composants (ports, esclaves, etc.) sont associés aux actions de communication selon lesquelles la trace de simulation est échantillonnée pour l'évaluation des assertions. Nous avons indiqué l'utilisation d'un fichier XML dit “d'instrumentation” qui permet d'établir le lien entre les composants concrets de la plateforme et les identifiants symboliques qui leur sont donnés dans l'assertion.

Dans le contexte RTL, l'échantillonnage se fait typiquement sur les fronts montants de l'horloge, voire également sur des communications TLM qui subsisteraient dans une plateforme partiellement concrétisée, c'est à dire hybride (TLM-RTL). Le fichier d'instrumentation est donc adapté en conséquence.

Parmi les expérimentations qui seront présentées dans le prochain chapitre, il existe une propriété similaire à *P9* (*c.f.* 6.5) mais pour une plateforme dans laquelle l'envoi des interruptions du DMA se fait en utilisant un appel de fonction TLM. Dans ce cas, si le raffinement ne touche pas cette fonction, l'assertion obtenue sera hybride et sa trace sera échantillonnée à la fois sur les fronts de l'horloge et sur les appels de la fonction de génération des interruptions (*c.f.* assertion 6.8).

Il nous faut donner la possibilité d'ajouter de nouvelles informations dans le fichier d'instrumentation. En effet, certains composants ne participent pas à l'échantillonnage de la trace, mais doivent néanmoins subir un léger traitement par ISIS. Ce sont les composants dont on doit simplement consulter la valeur dans certains points de la trace, mais qui ne seraient pas nécessairement équipés d'une méthode C++ d'accès à la valeur (communément appelée “*getter*”). C'est le cas, dans l'assertion 5.4, des signaux `cpu_HWRITE` et `cpu_HADDR` par exemple.

Une balise “<needed>” permet de les spécifier dans le fichier d'instrumentation, de façon à ce que tous les getters nécessaires pour autoriser la consultation de leur valeur depuis l'extérieur de la plateforme (où se trouvent les moniteurs) soient générés par ISIS.

Une balise “<needed_rose_fell>” permet quant à elle de produire en plus les fonctions “**rose**”, “**fell**” et “**prev**” mentionnées ci-dessus.

Le programme 5.5 montre un extrait de la nouvelle DTD⁶ du fichier d'instrumentation. Conformément à cette définition, le fichier d'instrumentation est composé d'un ensemble de liens “**link**” et optionnellement d'un ensemble d'éléments du type “**needed**” et “**needed_rose_fell**”. Comme les éléments du type “**link**”, les deux nouveaux éléments sont composés d'une balise “**component**” et d'une balise “**variable**”.

```
<!ELEMENT links (link+,needed*,needed_rose_fell*)>
  <!ELEMENT link (component,variable)>
  <!ELEMENT needed (component,variable,privatefunctioncalled?)>
  <!ELEMENT needed_rose_fell (component,variable)>
    <!ELEMENT component (#PCDATA)>
    <!ELEMENT variable (#PCDATA)>
  ...
```

PROGRAMME 5.5 – Extrait de la nouvelle DTD du fichier d'instrumentation d'ISIS

Comme nous l'avons décrit dans la Figure 5.1, le fichier d'instrumentation est automatiquement généré pour chaque propriété raffinée.

⁶*Document Type Definition*, c'est un langage qui permet de définir la structure des éléments du document XML. <http://www.w3schools.com/DTD/default.asp>

5.3 L'implémentation des SEREs

Dans cette section, nous allons revenir sur l'implémentation des opérateurs SERE dont l'utilité a été mentionnée dans le précédent chapitre (*c.f.* section 4.3.2). Comme les autres opérateurs PSL implémentés dans ISIS et Horus, l'automatisation de la génération des moniteurs correspondants à ces opérateurs nécessite au préalable que les moniteurs élémentaires implémentant la sémantique des opérateurs PSL soient définis et intégrés dans la bibliothèque des moniteurs. Ensuite, le schéma d'interconnexion de ces moniteurs doit être défini afin de constituer automatiquement le moniteur complet.

5.3.1 Définition des moniteurs SERE élémentaires

L'implémentation des moniteurs SERE dans ISIS est fortement inspirée de celle des moniteurs SERE dans Horus[MAGB07]. Nous allons alors commencer par introduire les principes de construction et d'interconnexion de ces derniers.

5.3.1.1 Moniteurs SERE dans Horus

La construction des moniteurs globaux se fait en interconnectant les opérateurs et les expressions booléennes. Les opérateurs sont les “connecteurs” qui implémentent la sémantique d'exécution. Les expressions booléennes sont représentées par des moniteurs booléens élémentaires.

L'évaluation de la formule PSL est conceptualisée par un jeton qui se propagerait du moniteur le plus à gauche vers le moniteur le plus à droite. Les connecteurs servent à déplacer le jeton d'un point d'évaluation à un autre. L'évaluation se fait au niveau des moniteurs des expressions booléennes.

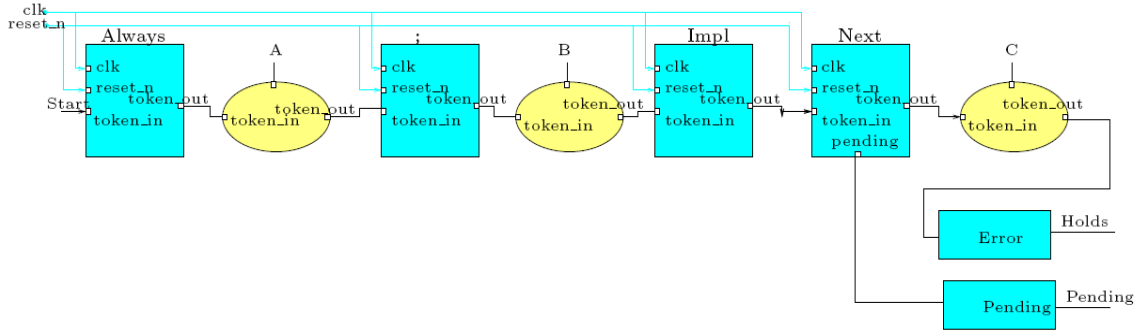
Le moniteur de la Figure 5.9 implémente la formule $P_1 = \text{always}(\{a;b\} \mid\!\!\rightarrow \text{next}(c))$. Elle est composée d'une implication suffixe reliant une SERE en partie gauche et une formule temporelle en partie droite. Dans la figure, les connecteurs sont représentés par des rectangles alors que les ovales représentent les moniteurs booléens.

Les interfaces de tous les moniteurs élémentaires sont identiques. En plus des entrées `reset_n` et `clk`, chaque moniteur possède une entrée “`token_in`” et une sortie “`token_out`”. Les moniteurs booléens possèdent en plus une entrée correspondant à l'expression booléenne.

Le moniteur global de la SERE produit deux sorties : *Pending* et *Holds*.

La sortie *Pending*='1' indique de l'assertion est à l'état **Pending**. La propriété est **violée** (*Fails*) quant la sortie *Holds*='0'. Elle est **satisfaite** (*Holds*) si la sortie *Holds*='1'. Si en plus la sortie *Pending*='0' alors elle est **fortement satisfaite** (*Holds Strongly*).

Comme nous l'avons souligné à travers l'exemple de la propriété P_7 (*c.f.* page 65), l'objectif de l'évaluation peut changer selon la position de la SERE dans la formule :

FIGURE 5.9 – Moniteur Horus de P_1 [MAGB07]

- L'évaluation des SEREs en partie gauche d'une implication suffixe vise à identifier les reconnaissances de l'expression régulière permettant de démarrer une nouvelle instance d'évaluation de la formule en partie droite. Si la SERE en partie gauche n'est pas reconnue, la formule en partie droite n'est pas évaluée.
- Pour les SEREs en partie droite, chaque instance d'évaluation vise en premier lieu à détecter les violations. Plusieurs instances d'évaluation peuvent évoluer simultanément dans une formule. Il est alors important d'identifier séparément chaque instance d'évaluation. Dans certains cas, une violation peut être masquée par une validation.

Pour implémenter ce mécanisme d'évaluation, chaque moniteur élémentaire existe en deux versions : une version dite monochrome pour les SERE en partie gauche et une version dite polychrome pour les SEREs en partie droite. Dans la première version, un seul jeton est propagé à la fois. Ce jeton est dit "monochrome". Dans la seconde version, plusieurs instances de jetons peuvent être traitées simultanément. Afin de les différencier, Horus utilise des jetons dits "colorés" ou "polychromes"[MAGB07].

La conception des connecteurs relatifs aux opérateurs de répétition est particulière. Ces connecteurs doivent permettre, à chaque cycle, d'itérer l'évaluation de leur opérande et de transmettre le résultat de l'évaluation courante au moniteur suivant. La Figure 5.10 est une illustration de l'interface du connecteur monochrome de l'opérateur "[*]" d' Horus. Ce connecteur doit permettre d'admettre une séquence vide. Ce chemin est représenté par la sortie "*bypass*" `token_out_bp`. Cette sortie indique au connecteur suivant de transmettre le jeton en entrée sans exécuter son traitement. Pour l'opérateur de composition séquentielle par exemple, ceci revient à transmettre le jeton en entrée vers la sortie au même cycle. Les entrées `op_token_out` et `op_token_out_bp` ainsi que la sortie `op_token_in` sont connectées à l'opérande :

- La sortie `op_token_in` permet d'itérer l'évaluation de l'opérande.
- L'entrée `op_token_out` contient le résultat d'évaluation de l'opérande.
- L'entrée `op_token_out_bp` n'est significative que si l'opérande peut reconnaître la

chaîne vide.

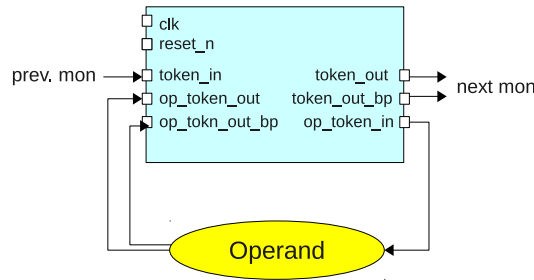


FIGURE 5.10 – Connecteur [*] monochrome

Dans le cas de la séquence {a;b[*];c}, par exemple, l’opérande est **b**, le connecteur précédent est le premier opérateur de composition séquentielle, le connecteur suivant est le second opérateur de composition séquentielle. La sortie *bypass* permet de reconnaître la trace a; c. La sortie “token_out” permet de reconnaître la trace a; b[+]; c.

5.3.1.2 Moniteurs SERE dans ISIS

Contrairement aux moniteurs élémentaires Horus décrits en VHDL (RTL), les moniteurs élémentaires dans ISIS sont des classes C++ et non pas des composants matériels synchrones. Selon le modèle d’observation implémenté, chaque classe hérite de la classe de base de tous les moniteurs nommée “Monitor” (*c.f.* section 3.2.2.3). Selon cette organisation, chaque classe doit offrir une surcharge de la méthode d’évaluation `update()`. Un exemple d’une classe C++ implémentant l’opérateur `before` est donné par le programme 5.6.

```
// monitor's interface
class mnt_before : public Monitor {
protected:
    //inputs
    bool reset_n, start, cond, expr;
    //outputs
    bool valid, checking, pending;
public:
    //Class constructor.
    mnt_before(const char * name,
               const OPType type = WEAK, //strong or weak version
               const OPDisc disc = EXCLUSIVE);
    //computation of evaluation result
    void update();
};
```

PROGRAMME 5.6 – l’interface du moniteur SystemC de l’opérateur before

Les classes implémentant les opérateurs SERE dans ISIS respectent la même organisation mais possèdent des interfaces différentes. Ainsi, un adaptateur sera nécessaire afin de pouvoir lier les moniteurs des opérateurs SERE aux moniteurs des opérateurs FL temporels.

Nous soulignons aussi que la méthode d'évaluation représentée dans 3.2.2.3 est la même pour les moniteurs SERE.

Par la suite, nous allons présenter les opérateurs SERE de composition séquentielle ";", l'implication suffixe non recouvrant "|=>" ainsi que la répétition "[*]"⁷. Comme pour les moniteurs SERE Horus, ces derniers sont implémentés en deux versions. Par souci de simplicité, nous présenterons dans un premier temps la version monochrome.

Il existe un traitement commun pour tous les opérateurs cités : le jeton en entrée doit être transmis vers la sortie au cycle d'évaluation suivant ou après un certain nombre de cycles. Dans Horus, ce fonctionnement est implémenté à l'aide d'un registre à décalage. Pour ISIS, un extrait de la classe effectuant ce même traitement est donné par le programme 5.7 :

```
class mono_shift{
    private :
        bool *checkbit;
        unsigned int size;
    public :

        mono_shift(unsigned int size_){
            size = size_;
            checkbit = new bool[size];
            for(unsigned int i=0;i<size;i++){
                checkbit[i] = false;
            }
        }
        //default constructor : one cycle shift
        mono_shift(){
            size = 2;
            ...
        }

        bool shift(bool d){
            // shift data
            for(unsigned int i=size-1;i>0;i--){
                {
                    checkbit[i] = checkbit[i-1];
                }
            }
        }
    }
```

⁷Les autres opérateurs implémentés ne seront pas présentés car ils peuvent être déduits à partir de ces derniers.

```

        // put new data
        checkbit[0] = d;
        //return last value
        return checkbit[size-1];
    }
    ...
}

```

PROGRAMME 5.7 – Implémentation logicielle d'un registre à décalage

Cette classe sera instanciée dans la plupart des opérateurs SERE implémentant un décalage d'un ou de plusieurs cycles (size-1).

Moniteur de composition séquentielle (concaténation)

Le programme 5.8 est un extrait de la classe C++ représentant le connecteur de composition séquentielle. Ce moniteur possède trois entrées :

- `reset_n` pour l'initialisation,
- `token_in` pour son activation,
- `token_in_bp` (*bypass*) pour transmettre le jeton directement vers la sortie. Cette entrée n'est active que si le moniteur précédent est un moniteur de répétition. Dans le cas contraire, elle est désactivée.

```

class mnt_concat : public Monitor
{
protected :
    // Inputs
    bool reset_n;
    /** trigger */
    bool token_in;
    /** baypass input. */
    bool token_in_bp;
    //output
    bool token_out;

private :
    mono_shift* token_shift;
    bool shift_out;
public :
    mnt_concat::mnt_concat(const char *n) : Monitor(n){ ... }
    void mnt_concat::update() {
        if(reset_n){ ... }
        else {
            // get des inputs :
            shift_out = token_shift->shift(token_in);
            token_out = shift_out || token_in_bp;
        }
    }
}

```

}

PROGRAMME 5.8 – Implémentation du moniteur de composition séquentielle ";"

La sortie "token_out" correspond à l'entrée "token_in_bp" si celle-ci est active, sinon, elle correspond à la sortie du registre à décalage "token_shift".

Moniteur de répétition [*]

L'implémentation de ce moniteur diffère du connecteur implémenté dans Horus. La Figure 5.11 fournit une vision structurelle de ces deux implémentations (moniteur ISIS à droite). Dans le moniteur Horus qui est un composant matériel synchrone, une bascule D est nécessaire pour couper la boucle structurelle. Le moniteur ISIS est en réalité une classe C++, mais nous représentons ici sa vision sous forme de composant matériel afin de faire le parallèle avec le moniteur Horus. Comme les moniteurs FLs temporels, la classe C++ de ce moniteur SERE est munie de la méthode "update()". Selon l'enchaînement suivi dans la méthode d'évaluation présentée dans la section 3.2.2.3, lorsque le moniteur global de la SERE est réévalué suite à la réception d'une notification, chaque appel à la méthode "update()" de ce moniteur correspondrait dans la figure 5.11 au calcul des nouvelles valeurs des sorties "token_out", "token_out_bp" et "op_token_in" du moniteur [*] en fonction des anciennes valeurs (chemin indiqué en gras et rouge sur la figure). Il n'y a pas de "boucle structurelle" au sens concret, et donc pas de composant assimilé à une bascule D dans le moniteur C++. L'évaluation correspondant au cycle suivant se fera, par construction, lors du prochain appel à la fonction "update()".

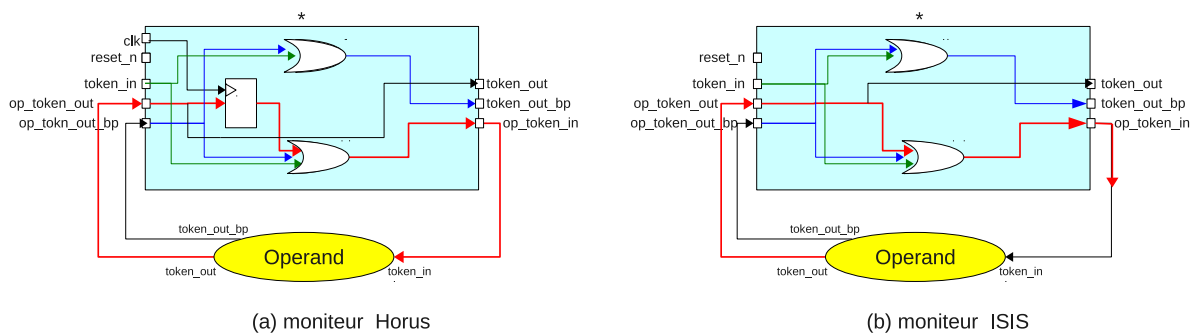


FIGURE 5.11 – Structure des connecteurs [*] Horus(à gauche) et ISIS(à droite)

Moniteur de l'implication suffixe non recouvrant

Identiquement au moniteur de concaténation, ce moniteur sert aussi à retarder, d'un cycle d'évaluation, la sortie du jeton en entrée. Le code associé est identique à celui du moniteur de composition séquentielle. La particularité de cet opérateur est qu'il peut être lié, en sortie, à une SERE polychrome ou à une formule FL temporelle. Dans le premier cas, un jeton coloré doit être généré. Dans le second cas, l'interface doit être adaptée

à celle que nous utilisons dans ISIS pour les FL, et les sorties “**valid**” et “**checking**” doivent être calculées à partir de la sortie “**token_out**” de ce moniteur. Ce traitement est réalisé par un adaptateur. La correspondance est la suivante :

- si **token_out** = '1' alors **valid** = '1' et **checking** = '1',
- si **token_out** = '0' alors **valid** = '1' et **checking** = '0'.

Dans le cas où l’opérande droit est une SERE le moniteur fait appel aux fonctionnalités d’un objet particulier appelé : “gestionnaire de jetons” ou “*token manager*”. Le principe de fonctionnement de cet objet est inspiré de son équivalent implémenté dans Horus. Il fera l’objet du prochain paragraphe.

Gestionnaire de jetons

Un jeton polychrome est associé à un ensemble de couleurs dont chacune correspond à une évaluation. Le gestionnaire de jetons est ajouté à la fin de chaque SERE polychrome. Il est responsable de la création et de la gestion des couleurs. Une couleur peut être dupliquée en entrée d’un opérateur de répétition. Elle peut aussi disparaître en sortie d’un moniteur booléen si ce booléen est faux.

La couleur est libérée si elle se trouve en sortie du moniteur global de la SERE c’est à dire que l’expression régulière a été reconnue. La séquence est alors validée et la couleur peut être réutilisée pour une prochaine évaluation.

L’évaluation échoue si la couleur disparaît en sortie de tous les moniteurs booléens. Dans ce cas aussi, la couleur est libérée et réutilisée.

Nous proposons d’expliquer le fonctionnement de cet objet sur l’exemple de la SERE : $P_2 : \text{always } (\{a\} \mid \Rightarrow \{b; \{c; d\}[*] ; e\})$. La Figure 5.12 donne un exemple d’une trace d’exécution. La création des jetons colorés se fait à la sortie du moniteur de l’implication suffixe. Cependant, pour simplifier l’explication, nous allons considérer les jetons colorés dès le début de la séquence.

Notons Ev_1 , Ev_2 , et Ev_3 les trois évaluations qui commencent respectivement aux instants #1 #3 #4. Les couleurs sont représentées par les formes : cercle, carré et triangle. À l’instant #3, Ev_1 n’est pas validée sur la trace $\{a; b; e\}$. Cependant, la couleur ne disparaît pas car elle a été dupliquée à l’instant #2, à la sortie de l’opérateur de composition séquentielle suivant b . Pour cette raison, Ev_1 n’échoue pas aux cycles #3 et 5. En effet, la couleur “cercle” est toujours présente au niveau du moniteur de répétition à ces instants. Cette évaluation est validée au cycle #7.

Ev_2 est validée à l’instant #7 sur la trace $\{a; b; c; d; e\}$. La couleur “carré” est alors libérée à cet instant. Elle est éliminée de tous les connecteurs polychromes.

Le gestionnaire de jetons doit pouvoir identifier la violation de Ev_3 survenue simultanément avec la validation de Ev_1 et Ev_2 ayant lieu à l’instant #7.

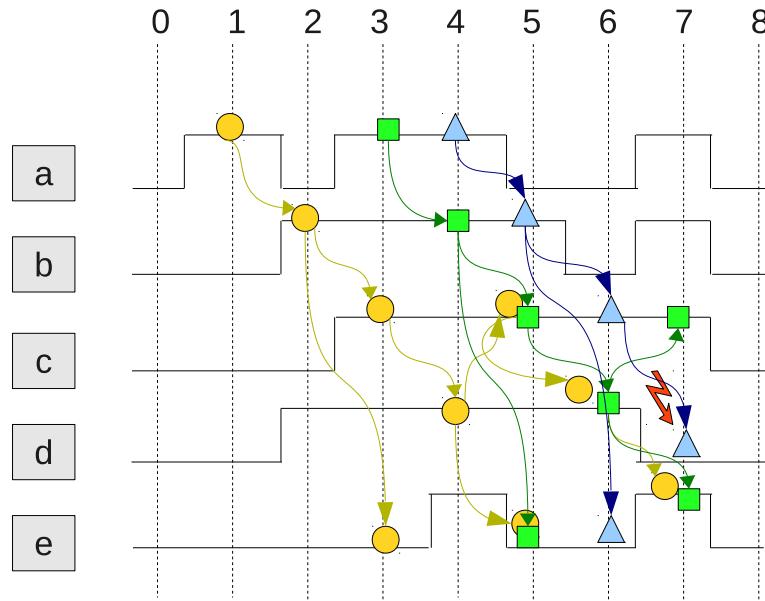
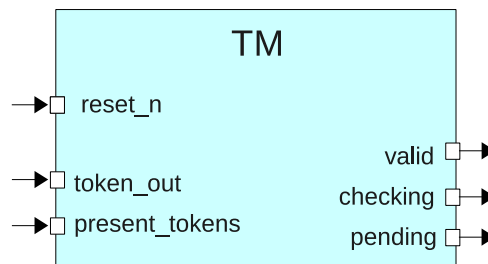
FIGURE 5.12 – Trace de la propriété P_2 

FIGURE 5.13 – Interface du gestionnaire des jetons

La Figure 5.13 est une schématisation structurale d'un objet TM de la classe “*token_manager*” qui implémente le gestionnaire de jetons. Cette classe définit un ensemble de fonctions parmi lesquelles nous citons en particulier la fonction “`create_new_token()`”. Cette dernière reçoit en entrée un jeton monochrome en provenance du connecteur de l'implication suffixe qu'elle convertit, en sortie, en une nouvelle couleur du jeton polychrome. Le gestionnaire garde une trace de toutes les couleurs qu'il a générées et qui ne sont pas encore libérées.

Pour la gestion des différentes couleurs, cet objet doit recevoir en entrée le jeton polychrome en sortie du dernier moniteur de la SERE par son entrée “`token_out`”. L'entrée “`present_tokens`” est la collection de toutes les sorties “`present_token`” de tous les connecteurs polychromes de la SERE. Ces sorties permettent de lui indiquer la présence d'une couleur au niveau du connecteur. Le gestionnaire fournit trois sorties “`valid`”, “`checking`” et “`pending`”.

Le calcul de l'état de l'évaluation Ev_i se fait selon les formules suivantes :

- Si `token_out` contient la couleur de l'évaluation i alors cette dernière est validée (`valid = '1'` et `checking = '1'`). La couleur est libérée.
- Si `token_out` ne contient pas la couleur de l'évaluation i alors :
 - Si cette couleur figure dans `present_tokens` alors l'évaluation correspondante est à l'état *Pending* (`valid = '1'`, `checking = '0'` et `pending = '1'`),
 - Si cette couleur ne figure pas dans `present_tokens` alors l'évaluation est violée (`valid = '0'`, `checking = '1'` et `pending = '0'`). La couleur est libérée.

5.3.2 Méthode d'interconnexion des SEREs

Il n'existe pas de contraintes syntaxiques (PSL_{ss}) imposées sur les opérateurs SERE (c.f. page 56). Ainsi, les opérandes d'un opérateur SERE peuvent être toutes composites (non booléens). Par conséquent la méthode d'interconnexion des opérateurs SERE est différente de celle des opérateurs FL car ceux-ci ne peuvent avoir, au plus, qu'un opérande composite. Afin d'expliquer ceci, nous allons présenter dans un premier temps, à travers un exemple de propriété, la méthode d'interconnexion des moniteurs FL.

Nous simplifions ici l'explication en la présentant avec une vision de composants matériels interconnectés structurellement, alors que les moniteurs ISIS sont logiciels.

5.3.2.1 Interconnexion des moniteurs FL élémentaires

Nous prenons comme exemple la propriété P_8 : **always** (`req ->next!` (`gnt before!` `req`)) dont l'arbre syntaxique a été montré dans la Figure 3.9 page 67. La méthode d'interconnexion permettant de construire le moniteur global consiste à instancier les moniteurs élémentaires en effectuant un parcours préfixe de l'arbre syntaxique de la propriété.

Dans le cas où l'un des opérandes du moniteur est composite, ce qui est le cas ici, pour l'opérateur d'implication logique et de l'opérateur "**next**", l'entrée "**expr**" du sous moniteur élémentaire est fixée à '1' car elle n'est pas pertinente, puisque c'est le sous moniteur suivant qui donnera le résultat de l'évaluation de cet opérande. Par exemple, on voit que l'entrée "**expr**" du moniteur "**implies**" est à '1' et que sa sortie "**checking**" est liée à l'entrée "**start**" du moniteur **next**.

Les sorties "**Checking**" et "**Valid**" du moniteur global correspondent aux sorties du dernier moniteur élémentaire dans la propriété. La sortie "**Pending**" correspond à la disjonction des sorties "**pending**" de tous les opérateurs forts de la formule.

5.3.2.2 Interconnexion des moniteurs SERE élémentaires

Dans le cas des SEREs, l'algorithme réalise un parcours infixe de l'arbre syntaxique. C'est un parcours en profondeur dans lequel le traitement de la racine est fait entre les appels sur les sous-arbres gauche et droit. Nous allons illustrer le résultat sur un exemple.

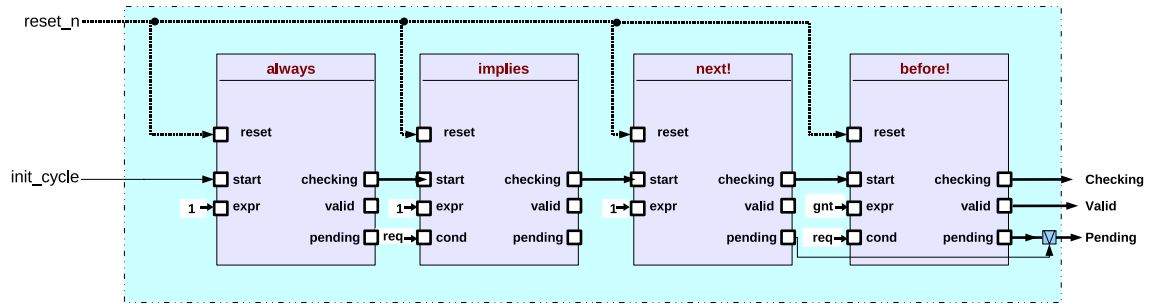


FIGURE 5.14 – Moniteur global généré par ISIS pour la formule FL P_8

La Figure 5.15 illustre l'arbre syntaxique et le moniteur associé à la séquence : $\{a;b\} \mid \Rightarrow \{c[*];d\}$. La construction du moniteur associé commence par la construction du fils gauche de la racine de l'arbre syntaxique (" $\{a;b\}$ "), suivi par celui de la racine (" $\mid \Rightarrow$ ") avant de passer ensuite à la construction du moniteur associé au sous arbre droit (" $\{c[*];d\}$ "). Le traitement est itéré sur les noeuds racine de chaque sous arbre.

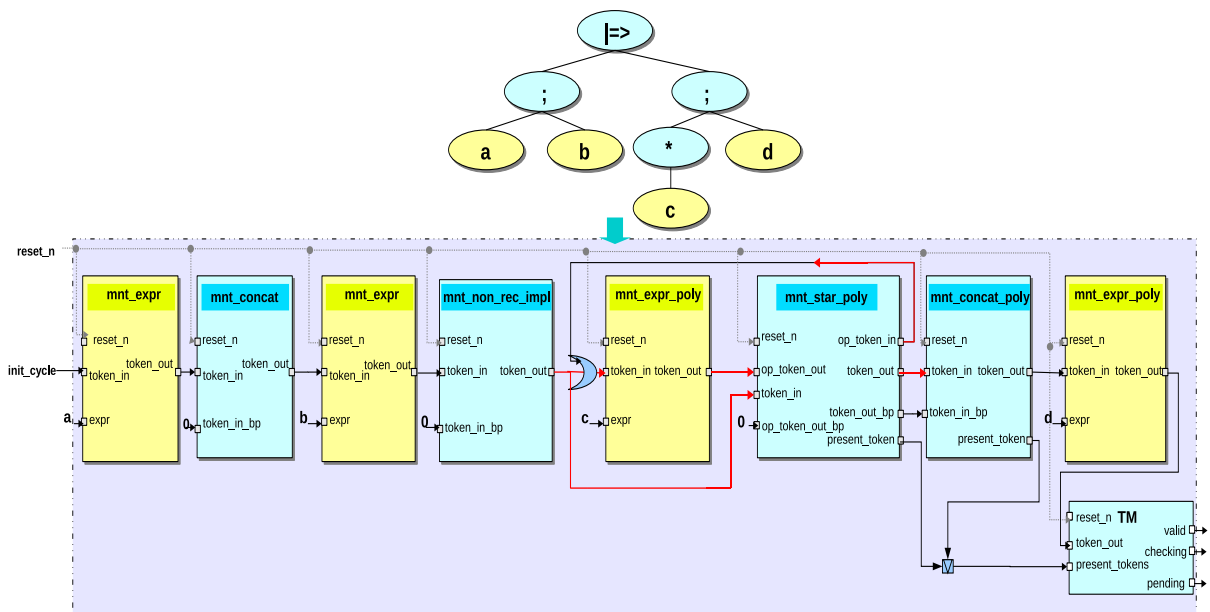


FIGURE 5.15 – Moniteur SERE global généré par ISIS

Il faut noter que la génération des jetons colorés n'est pas représentée dans la Figure 5.15. La fonction de génération est invoquée en sortie du connecteur d'implication suffixe. La séquence en partie droite de cet opérateur est implémentée en utilisant la version polychrome des moniteurs. Les sorties des moniteurs des connecteurs polychromes sont dirigées vers l'entrée du générateur de jetons (TM).

Les explications sur le moniteur de répétition “[*]”, données dans la section 5.3.1.2 se concrétisent ici de la façon suivante : le composant identifié par “*Operand*” dans la Figure 5.11 est ici le moniteur du type “`mnt_expr_poly`” (moniteur booléen polychrome) de l’opérande “`c`”. Selon la méthode

d'évaluation adoptée, l'appel à la fonction “`update()`” du moniteur global permettra le calcul des nouvelles valeurs des sorties “`token_out`” et “`op_token_in`” du moniteur “`mnt_star_poly`”, en prenant en compte la valeur de “`c`”. La mise à jour de l'entrée “`token_in`” n'aura lieu qu'au prochain appel à la fonction “`update()`”.

L'entrée “`op_token_out_bp`” est maintenue à '0' ici parce que le moniteur “`mnt_expr_poly`” associé à “`c`” ne possède pas de sortie “`token_out_bp`”.

Le raisonnement est identique pour l'opérateur “`[+]`”.

Bilan

Nous avons présenté à travers ce chapitre les principales fonctionnalités implémentées afin de fournir aux concepteurs les outils nécessaires pour permettre d'inclure les activités de vérification dans le flot de développement des SoCs complexes. Le principal apport est la proposition d'un outil prototype pour le raffinement automatique des assertions transactionnelles. Son but est de faciliter la réutilisation des assertions transactionnelles aux niveaux les moins abstraits et d'encourager par conséquent l'adoption des méthodes de l'ABV dès les premières étapes de conception.

L'outil proposé est conçu de façon à permettre le raffinement incrémental des propriétés. Les assertions générées peuvent être hybrides ou au niveau RTL.

Des extensions ont été introduites à l'outil ISIS afin de fournir le support nécessaire à la vérification dynamique des assertions raffinées hybrides et RTL.

Dans le chapitre qui suivra, nous allons présenter les résultats de l'utilisation de l'outil prototype de raffinement sur un ensemble varié de cas d'étude.

Chapitre 6

Expérimentations

Sommaire

7.1	Contributions	171
7.2	Travaux futurs	173
7.2.1	Travaux à court terme : génération des chronogrammes	173
7.2.2	Perspectives	173

“The good news about computers is that they do what you tell them to do. The bad news is that they do what you tell them to do.”

Ted Nelson

Introduction

Dans ce chapitre nous allons étudier les résultats de l’application de notre solution de raffinement. Les concepts développés tout au long des deux chapitres précédents ont été appliqués à des modèles de tailles et de caractéristiques variées : mélange de canaux primitifs et hiérarchiques, sans ou avec notion de temps de simulation, utilisation ou pas de la bibliothèque TLM pour SystemC, raffinement au niveau RTL ou hybride.

Des expérimentations sur deux plateformes sont relatées par la suite. Pour chaque cas d’étude, nous commencerons par la présentation de la plateforme au niveau transactionnel ainsi que de quelques propriétés caractéristiques au niveau transactionnel. Ensuite, suivra la description de la plateforme raffinée et du protocole utilisé, ainsi que les propriétés obtenues en utilisant l’outil de raffinement. Une analyse des différences entre les propriétés d’origine et les propriétés raffinées sera faite. Les temps CPU pour les différentes simulations (sans et avec surveillance, propriétés transactionnelles, RTL ou hybrides), sera faite dans la section 6.3.

6.1 Cas d’étude : Plateforme de décodage vidéo Motion—JPEG

6.1.1 Plateforme et assertions transactionnelles

Le premier cas d’étude est issu des travaux de [GGP08]. C’est une plateforme MP-SoC¹ décrite au niveau transactionnel (*Transaction Accurate*) et codée en SystemC. Cette plateforme permet une exécution native d’une application de décodage de vidéos MJPEG. La liaison entre les deux domaines (logiciel et matériel) est faite par l’intermédiaire d’une couche dite “*Hardware Abstraction Layer* (HAL)” permettant d’abstraire les détails physiques du matériel par rapport au logiciel. Le logiciel est implémenté comme un ensemble de processus POSIX². Nous nous intéressons uniquement à la plateforme matérielle. La Figure 6.1 illustre l’architecture de cette plateforme.

La plateforme matérielle est composée d’un nombre configurable d’unités d’exécution «EU», d’une mémoire ROM «Memory», d’un sémaphore matériel «SEMRAM» et d’un

¹Multi-Processor SoC

²C’est un standard IEEE (Std 1003.1c) qui définit des interfaces de programmation des variantes d’UNIX

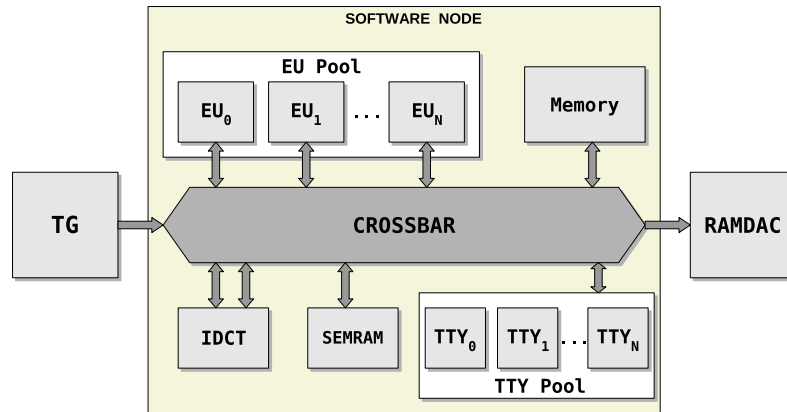


FIGURE 6.1 – Plateforme MJPEG (source [GGP08])

ensemble de terminaux «TTY»(un terminal par unité d'exécution). Elle contient aussi deux périphériques : le générateur de trafic (TG) qui prend en entrée une vidéo MJPEG, et un *buffer* d'affichage vidéo nommé RAMDAC³ pour l'affichage de la vidéo décodée. La fonction de décodage est réalisée par l'IDCT (*Inverse Discrete Cosine Transform*). Les composants sont reliés par un bus à mémoire mappée transactionnel.

Deux propriétés PSL ont été spécifiées afin de surveiller le bon acheminement des données à destination du RAMDAC. La première vérifie l'intégrité des données envoyées du processeur vers le RAMDAC. La deuxième vérifie l'absence de perte de données transmises via le canal de communication.

Nous travaillons ici sur une plateforme contenant une seule unité d'exécution, les transferts sur le bus ne sont pas pipelinés (c'est à dire qu'un seul transfert au plus y est réalisé à tout moment).

6.1.1.1 P_1 : Intégrité des données

La première propriété spécifie le comportement suivant : “*Les données écrites dans le RAMDAC doivent correspondre aux données émises par l'unité d'exécution.*”. La spécification formelle de cette propriété en PSL donne l'assertion décrite par le programme 6.1. La couche de modélisation indique qu'à chaque fois que le processeur EU_0 appelle la fonction d'écriture `write` à destination du RAMDAC, la donnée écrite (deuxième paramètre, noté p2) est mémorisée dans la variable auxiliaire `req_data`. La prochaine fois que la fonction d'écriture du RAMDAC sera appelée, la donnée écrite devrait correspondre à la donnée mémorisée dans `req_data`.

```
vunit tlm_proprdac {
    // The data sent by the EU are correctly received by the RAMDAC.
    // HDL_DECLS :
    unsigned int req_data;
```

³Random Access Memory Digital-to-Analog Converter


```

bool for_rdac;
memory_map ** mem_map;

// HDL_STMTs :
if( eu.write_CALL() ) {
    mem_map = noc.get_memory_map();
    // test if the request is sent to the RAMDAC
    if(eu.write.p1 >= mem_map[6]->base_addr &&
        eu.write.p1 <= mem_map[6]->end_addr){
        for_rdac = true;
        req_data = eu.write.p2;
    }else for_rdac = false;
}

// PROPERTY :
assert
always ( eu.write_CALL() && for_rdac
        ->next_event(rdac.write_CALL())(
            req_data == rdac.write.p2 )
);

```

PROGRAMME 6.1 – Propriété P_1 MJPEG transactionnelle

Le processeur peut envoyer des requêtes à destination de tous les composants esclaves connectés au canal (RAMDAC, TTY, IDCT, etc.). Pour cette raison, dans la couche modélisation de l'assertion, un traitement est fait afin de positionner à vrai la variable booléenne `for_rdac` (et de sauvegarder la donnée dans `req_data`) uniquement si la requête d'écriture émise par le processeur vers le canal (nommé "noc" dans l'assertion et identifié dans le fichier d'instrumentation, voir annexe C.1) est à destination du RAMDAC et non pas d'un autre composant. La configuration de l'espace d'adressage (*memory map*) est faite dynamiquement au début de chaque exécution. L'appel de la fonction `get_memory_map()` du canal permet de récupérer la structure de données `mem_map` contenant la configuration actuelle de l'espace d'adressage. Le RAMDAC correspond à l'entrée numéro 6 dans l'espace d'adressage, son espace réservé est compris entre l'adresse de début contenue dans le champ `base_addr` et l'adresse de fin donnée par `end_addr`. Dans la propriété, à chaque appel de la fonction d'écriture du processeur `EU_0`, l'adresse contenue dans le premier paramètre est comparée par rapport à l'espace d'adressage réservé au RAMDAC. Le résultat de la comparaison est sauvegardé dans la variable booléenne `for_rdac`.

La trace de la propriété est échantillonnée sur les appels de la fonction `write` du composant RAMDAC et de la fonction `write` du processeur. Le fichier d'instrumentation associé à cette assertion est donné en annexe (*c.f.* programme C.1). Un extrait de la trace d'exécution de la plateforme instrumentée avec le moniteur généré à partir de cette

assertion est décrit en annexe.

6.1.1.2 P_2 : Non perte des données

La seconde propriété spécifie que “*Le nombre d’écritures faites dans le RAMDAC doit toujours correspondre au nombre de requêtes d’écriture faites par l’unité d’exécution*”. Dans cette assertion, deux compteurs globaux doivent être définis dans la couche de modélisation. Le premier “`req_count`” est destiné à calculer le nombre de requêtes d’écriture émises par l’unité d’exécution EU_0 à destination du RAMDAC. Le second “`write_count`” sert à calculer le nombre d’appels à la fonction d’écriture du RAMDAC. À chaque fois que le RAMDAC exécute une requête d’écriture, il faut que les valeurs des deux compteurs soient égaux. L’assertion PSL décrivant cette propriété est (*c.f.* programme 6.2) :

```
vunit tlm_proprdac2 {
    // The communication is lossless in the number of write operations.
    // HDL_DECLS :
    unsigned int req_count = 0;
    unsigned int write_count = 0;
    memory_map ** mem_map;
    // HDL_STMTs :
    if( eu.write_CALL() ) {
        //get the memory_map
        mem_map = noc.get_memory_map();
        if(eu.write.p1 >= mem_map[6]->base_addr &&
            eu.write.p1 <= mem_map[6]->end_addr){
            req_count++;
        }
    }
    if( rdac.write_CALL() ) { write_count++; }
    // PROPERTY :
    assert
    ( always (rdac.write_CALL() -> (req_count == write_count))
    @ (eu.write_CALL() || rdac.write_CALL()));
}
```

PROGRAMME 6.2 – Propriété P_2 MJPEG transactionnelle

Le même traitement est effectué afin de s’assurer que la requête est destinée au RAMDAC avant d’incrémenter le compteur des requêtes. On note aussi que puisque la fonction booléenne `eu.write_CALL` n’apparaît que dans la couche modélisation, l’opérateur `@` est utilisé afin de l’ajouter explicitement à la liste de sensibilité du moniteur de l’assertion. Le fichier d’instrumentation associé à cette assertion est identique à celui de la première assertion. Un extrait de la trace d’exécution de la plateforme instrumentée avec le moniteur associé à cette assertion est donnée en annexe.

6.1.2 Plateforme et assertions RTL

Nous avons raffiné cette plateforme au niveau RTL en implémentant le protocole Wishbone[ope10]. Concrètement, nous avons remplacé le bus transactionnel par un bus Wishbone. Nous n'avons pas réalisé les implémentations correspondantes des autres composants, mais simplement introduit des transacteurs (*c.f.* page 48) Wishbone sur leurs interfaces. Les composants jouant le rôle de maître (EU) ont été reliés à des *driver* tandis que les composants jouant le rôle d'esclave (MEMORY, SEMMEM, RAMDAC, TTY, IDCT, TG) ont été reliés à des *responder*.

Dans les fichiers de configuration de la plateforme d'origine, les auteurs ont spécifié que la fréquence de fonctionnement des processeurs est de 100 Mhz. Nous avons alors spécifié une horloge globale dont la période est équivalente à cette fréquence de fonctionnement soit 10 ns.

6.1.2.1 Norme Wishbone

La Figure 6.2 schématise les interfaces au niveau signal du *driver* et du *responder* implémentés. Les signaux affichés permettent d'effectuer des transferts simples. Pour les rafales, des signaux supplémentaires sont nécessaires. Conformément à la norme Wishbone, chaque composant (maître ou esclave) possède les entrées de synchronisation `clk`, `reset_n`, deux ports pour les données entrantes (`DAT_I`) et sortantes (`DAT_O`). Le port `SEL_I/O` permet d'indiquer l'emplacement des données dans le bus des données.

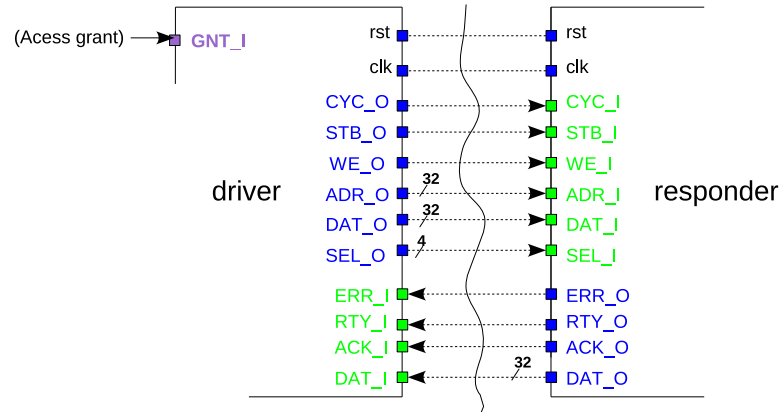
Le *responder* possède les ports `CYC_I`, `STB_I` et `WE_I` permettant d'indiquer, respectivement :

- l'existence d'une requête émise par le maître (`CYC_I='1'`),
- le début effectif de la requête (`STB_I='1'`),
- la nature de la requête : écriture (`WE_I='1'`) ou lecture.

Le *driver* possède trois ports indiquant la fin de la transmission : `ACK_I` signifie que le transfert s'est terminé correctement, `ERR_I` que le transfert a échoué, et `RTY_I` que le transfert n'a pu débuter car l'esclave n'était pas prêt à recevoir une requête. Les deux derniers sont optionnels. Les interfaces maître et esclave sont totalement symétriques. À chaque port de sortie `P_O` correspond un port d'entrée `P_I`.

Si la plateforme comporte plusieurs maîtres connectés à un même bus, un protocole d'arbitrage doit être implémenté afin de gérer les accès concurrents au bus. Les maîtres expriment leur besoin d'accéder au bus par l'intermédiaire du signal `CYC_O`. Le maître ayant accès au bus reçoit un signal `GNT_I` actif. Ainsi, chaque maître doit posséder en plus une entrée `GNT_I` lui indiquant s'il a accès au bus.

Dans le cas où un maître peut communiquer avec plusieurs esclaves, l'identification de l'esclave concerné se fait à l'aide d'une fonction de décodage d'adresses. La sélection de

FIGURE 6.2 – Interfaces *driver*, *responder* du Wishbone

l'esclave cible est faite en utilisant le signal `STB_I` en sortie du Bus. À l'exception de ce signal, tous les signaux sortant du bus sont partagés par tous les composants esclaves.

Le transfert sur le bus respecte le protocole “poignée de main” (*handshake*). Comme le montre la Figure 6.3, le maître commence son transfert par l'activation du signal `STB_O`. Ce signal reste actif jusqu'à la réception de l'un des signaux de fin de transfert. À la réception de ce signal, le maître désactive sa sortie `STB_O`.

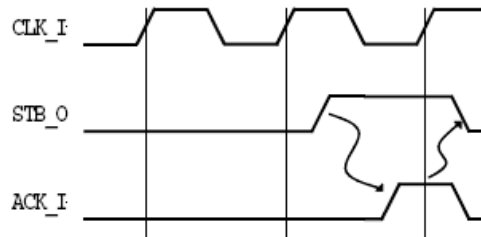


FIGURE 6.3 – Protocole “poignée de main” du Wishbone (source :[ope10])

6.1.2.2 Raffinement des assertions

Nous avons fait appel à l'outil prototype présenté au précédent chapitre afin de raffiner les assertions PSL énoncées. Dans les deux propriétés, les deux fonctions de communication observées ont été raffinées. Les détails du processus de raffinement sont décrits en annexe (*c.f.* section C.2).

P_1 : Intégrité des données

La Figure 6.4 décrit l'équivalent du chronogramme associé à la fonction d'écriture du processeur `EU_0`. Elle montre que le démarrage de la requête d'écriture du processeur est marqué par le passage du signal `eu0_STB` à '1' tout en ayant `eu0_WE` et `eu0_CYC` à '1'. Cette indication se traduira dans la propriété par un appel à la fonction `rose`. Ces sig-

Signals Evolution for method :		write	Clock:	clk.POSEdge
signal	cycle	0	1	
top_inst_0/eu0_CYC			1	
top_inst_0/eu0_WE			1	
top_inst_0/eu0_STB		0	1	
top_inst_0/eu0_ADDR			x	
req_data			x	
base_addr			x	
port			x	
for_rdac			x	
write start condition		eu0_STB==1		

FIGURE 6.4 – Chronogramme de la méthode write du processeur

naux marquent le début de l'opération d'écriture et correspondent au point d'observation "CALL" au niveau transactionnel.

Selon la figure, tous les signaux de contrôle, d'adresse et de données correspondant à l'opération d'écriture simple et à ses paramètres sont envoyés au même cycle d'horloge. Par conséquent, aucune contrainte temporelle n'est générée. Le raffinement est uniquement structurel.

Le chronogramme décrivant l'évolution des signaux correspondant à la méthode `write` du RAMDAC est semblable à celui décrit dans la Figure 6.4. L'appel à la fonction `write` correspond principalement à la sélection du RAMDAC (passage de la valeur du signal entrant `rdca_STB` de '0' à '1') et à l'envoi de la commande indiquant une écriture (`rdac_WE = '1'`).

L'assertion PSL donnée dans le programme 6.3 représente la sortie du processus de raffinement pour la première propriété. Les deux fonctions de communication ont été raffinées. L'assertion résultante est alors au niveau RTL et l'observation se fait uniquement sur le front montant de l'horloge `clk`. Le fichier d'instrumentation généré est donné en annexe (programme C.2).

```

vunit rtl_proprdac {
    // The data sent by the EU are correctly received by the RAMDAC.
    // HDL_DECLs :
    unsigned int req_data;
    bool for_rdac;
    memory_map ** mem_map;
    // HDL_STMTs :
    if( eu_CYC && eu_WE && rose(eu_STB,clk) && clk.rising() ) {
        //get the memory map
        mem_map = noc.get_memory_map();
        if(eu_ADR >= mem_map[6]->base_addr &&
            eu_ADR <= mem_map[6]->end_addr){
            for_rdac = true;
            req_data = eu_DAT;
        }
    }
}

```

```

        }else for_rdac = false;
    }
// PROPERTY :
assert
(
    always(
        ( eu_CYC && eu_WE && rose(eu_STB,clk) && for_rdac
          -> next_event( rdac_CYC && rose(rdac_STB,clk) && rdac_WE )
                    (req_data == rdac_DAT)) )
)@(clk.rising());
}

```

PROGRAMME 6.3 – Propriété P_1 MJPEG raffinée

Un extrait de la trace d'exécution de la plateforme RTL instrumentée avec le moniteur de cette assertion est commenté en annexe (*c.f.* annexe C.2.3).

P_2 : Non perte des données Le programme 6.4 contient l'assertion résultant du raffinement de la deuxième propriété. Les fonctions raffinées sont identiques à celles de l'assertion précédente. Nous soulignons que dans les deux cas, aucune règle de transformation n'a été appliquée car aucune contrainte temporelle n'a été déduite à partir des chronogrammes.

```

vunit rtl_proprdac2 {
// The communication is lossless in the number of write operations.
// HDL_DECLs :
    unsigned int req_count = 0;
    unsigned int write_count = 0;
    memory_map ** mem_map;
// HDL_STMTs :
    if( eu_CYC && eu_WE && rose(eu_STB,clk) && clk.rising() ) {
        //get the memory map
        mem_map = noc.get_memory_map();
        if(eu_ADR >= mem_map[6]->base_addr &&
           eu_ADR <= mem_map[6]->end_addr){
            req_count++;
        }
    }
    if( rdac_CYC && rose(rdac_STB,clk) && rdac_WE && clk.rising() )
    { write_count++; }

// PROPERTY :
assert
(
    always(
        (rdac_CYC && rose(rdac_STB,clk) && rdac_WE
          -> (write_count == req_count) )
    )
)

```

```
)@(clk.rising());
```

PROGRAMME 6.4 – Propriété P_2 MJPEG raffinée

Le fichier d’interconnexion de cette assertion est semblable à celui de la première assertion. Un extrait de la trace d’exécution est donné en annexe (*c.f.* annexe C.2.3).

6.2 Cas d’étude : Plateforme de compression d’images

6.2.1 Plateforme et assertions transactionnelles

La plateforme traitée est un modèle développé par Astrium Toulouse. Il s’agit d’une application de compression spectrale d’images comme cas d’étude du projet “SoCKET”⁴. L’architecture illustrée dans la Figure 6.5 est la deuxième version d’une architecture expérimentale étudiée par Astrium. Elle est décrite au niveau transactionnel et codée en SystemC-TLM2. La plateforme matérielle est composée de trois canaux de communication mappés, identiques interconnectés et d’un ensemble de composants. Le premier canal (*bus_a*) permet la transmission des données entre un pseudo processeur (*Leon_a*), un contrôleur DMA (*DMA_a*) et une mémoire (*Mem_a*). Le canal *bus_b* met en relation le pseudo processeur *Leon_b*, le contrôleur *DMA_b*, la mémoire *Mem_b* et un composant nommé FFT effectuant une transformée de Fourier rapide sur l’image. Le canal *bus_io* est relié au composant IO constituant le point d’entrée et de sortie de l’image traitée. La taille de l’image en sortie est divisée par 10 par rapport à la taille de l’image en entrée. Nous précisons aussi les caractéristiques suivantes :

- Les ports *initiator* sont du type `simple_initiator_socket` et les ports *target* sont du type `simple_target_socket`.
- Les bus sont de taille 32 octets.
- Le protocole implémenté est le protocole de base de TLM.
- Les transports sont bloquants (*b_transport*).
- Des latences (*wait*) sont introduites au niveau de chaque composant afin de simuler le temps de traitement et de transmission des données.

Dès qu’une nouvelle image arrive au niveau de l’IO, ce dernier envoie une interruption en direction du *Leon_a*. Le processeur programme le *DMA_a* afin de copier l’image vers *Mem_a*. Le processeur effectue un sous-échantillonnage afin de conserver les données sans défaut. La taille de l’image est alors divisée par deux. Le *Leon_a* configure le *DMA_b* afin de copier l’image résultante vers *Mem_b*. Le *Leon_b* configure le bloc FFT pour faire la transformée de Fourier sur l’image. À la réception de l’interruption de la FFT, le *Leon_b*

⁴SoC toolKit for critical Embedded sysTems. <http://socket.imag.fr/>

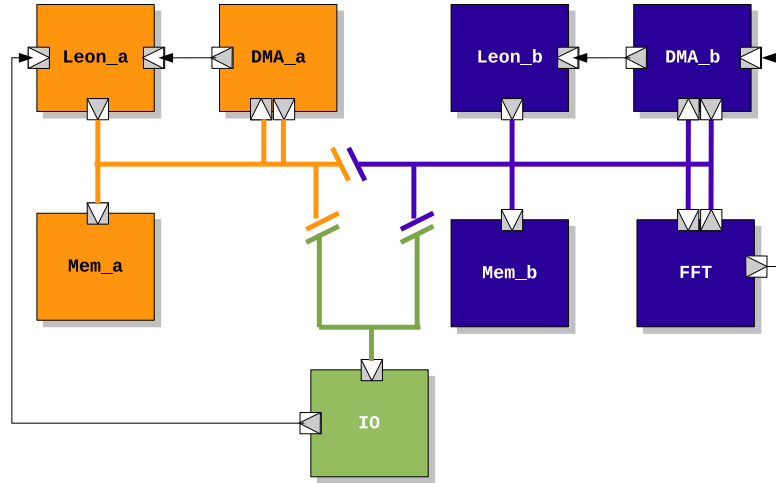


FIGURE 6.5 – Plateforme de compression d'images AstriumV2

effectue la compression de l'image. Il configure le DMA_b pour copier l'image compressée vers le IO. Une interruption est aussi envoyée à la fin de ce traitement. La configuration des composants nécessite un seul appel à la fonction de transport. Cependant, plusieurs appels peuvent être nécessaires lors des transferts d'images.

Nous considérons trois propriétés spécifiées par les auteurs de la plateforme. Ces propriétés visent à vérifier le fonctionnement du logiciel vis-à-vis du matériel. La première propriété concerne particulièrement la vérification de l'interaction avec le DMA_a et exprime, par conséquent, un besoin similaire à celui de la propriété P_9 de la plateforme `pv_dma`. La seconde concerne la vérification du fonctionnement de la FFT et la troisième vérifie la non perte des données à l'entrée du modèle.

6.2.1.1 P_3 : Configuration du DMA_a

La première propriété modélise le comportement suivant : “*le DMA_a ne doit pas être reconfiguré tant qu'il n'a pas fini son transfert courant*”. Selon l'implémentation de la plateforme, le début du transfert est marqué par la configuration du registre de contrôle du DMA_a par le Leon_a : l'écriture de la valeur de `dma_reg::dma_control_start_mask` (égal à 1) dans le registre `dma_reg::a_dma_control`. La formalisation de cette propriété donne l'assertion décrite par le programme 6.5.

```
vunit prop3 {
  // no DMA_a configuration before the end of the current transfer
  // HDL_DECLS
  const unsigned long ctrl_reg = 16777216 + dma_reg::a_dma_control;
  const unsigned int start_transfer = dma_reg::dma_control_start_mask;
  unsigned char* data_ptr;
  unsigned int value;

  // HDL_STMTs :
```



```

if(leon_a.write_block_CALL())
{
    data_ptr = leon_a.write_block.p2; // data buffer
    value = data_ptr[0] + (data_ptr[1] << 8) + // value written
           (data_ptr[2] << 16) + (data_ptr[3] << 24);
}

// PROPERTY :
assert
    always
        ( (leon_a.write_block_CALL() &&
            leon_a.write_block.p1 == ctrl_reg &&
            value == start_transfer)
          -> next (dma_a.generate_irq_CALL()
                  before! (leon_a.write_block_CALL() &&
                          leon_a.write_block.p1 ==
                          ctrl_reg &&
                          value == start_transfer)
                )
        );
}

```

PROGRAMME 6.5 – Propriété P_3 au niveau transactionnel

La trace de l’assertion est échantillonnée sur les appels de la fonction “`write_block()`” du `Leon_a` et la fonction “`generate_irq()`” du `DMA_a`. La première possède trois paramètres : l’adresse de la transaction (`p1`), un pointeur vers les données à écrire (`p2`) et la taille des données. Ici, la couche modélisation est utilisée pour déclarer deux variables et deux constantes. Les constantes “`ctrl_reg`” et “`start_transfer`” contiennent respectivement l’adresse du registre de contrôle du `DMA_a` et la valeur associée à `START`. Un traitement est nécessaire dans la couche de modélisation afin de récupérer les données transmises depuis le pointeur en paramètre de la fonction “`write_block()`”. Ce traitement est emprunté au code de la fonction de transport. La variable “`data_ptr`” permet de récupérer le pointeur vers les données. La variable “`value`” contient le résultat du calcul de la valeur des données.

La fonction “`generate_irq()`” permet d’envoyer une interruption vers le `leon_a`. Le port *initiator* permettant l’envoi de la transaction d’interruption est directement relié à l’un des ports *target* du `Leon_a`.

Le fichier d’instrumentation de cette propriété est fourni en annexe (*c.f.* programme D.1) suivi par un extrait de la trace d’exécution de la plateforme instrumentée avec cette propriété (*c.f.* trace D.1.1).

6.2.1.2 P_4 : Configuration de la FFT

La propriété P_4 consiste à vérifier que “la FFT ne doit pas être reconfigurée avant la fin de son traitement”. Elle indique qu’à chaque fois que le `Leon_b` configure le registre `fft_reg::read_addr` de la FFT, celle-ci doit finir son traitement avant sa prochaine configuration. La formalisation de cette propriété donne l’assertion PSL décrite dans le programme 6.6.

Comme pour les autres propriétés, il est impératif d’identifier les traitements relatifs au “début” et à la “fin” de chaque action afin de pouvoir formaliser la propriété.

À la fin de la configuration de la FFT, le `Leon_b` écrit dans le registre `fft_reg::a_read_length` la taille de l’image à échantillonner. Tout au long de son traitement, la FFT met à jour la taille de l’image dans son registre `fft_reg::a_write_length`. Le traitement se termine quand le `Leon_b` lit dans ce registre la valeur de `fft_size`.

```
vunit prop4 {
    // good FFT configuration
    // HDL_DECLs :
    const unsigned long fft_address = 33554432;
    const unsigned long a_read_addr = fft_address +
                                     fft_reg::a_read_addr;
    const unsigned long a_read_length = fft_address +
                                       fft_reg::a_read_length;
    const unsigned long a_write_length = fft_address +
                                       fft_reg::a_write_length;

    unsigned char* write_data_ptr;
    unsigned char* read_data_ptr;
    unsigned int fft_size;
    unsigned int write_length_value;

    // HDL_STMTs :
    //configure the FFT register with the packet length
    if(leon_b.write_block_CALL() && leon_b.write_block.p1 ==
        a_read_length)
    {
        write_data_ptr = leon_b.write_block.p2;
        fft_size = write_data_ptr[0] + (write_data_ptr[1] << 8) +
                  (write_data_ptr[2] << 16) + (write_data_ptr[3] << 24);
    }
    //read the packet length on the FFT register
    if(leon_b.read_block_END() && leon_b.read_block.p1 == a_write_length)
    {
        read_data_ptr = leon_b.read_block.p2;
        write_length_value = read_data_ptr[0] + (read_data_ptr[1] << 8)
                           + (read_data_ptr[2] << 16) + (read_data_ptr[3] << 24);
    }
}
```

```

// PROPERTY :
assert
  always (leon_b.write_block_CALL()
    && leon_b.write_block.p1 == a_read_addr
    -> next ( (leon_b.read_block_END()
      && leon_b.read_block.p1 == a_write_length
      && write_length_value == fft_size)
      before! (leon_b.write_block_CALL()
        && leon_b.write_block.p1 == a_read_addr
      )
    )
  );
}

```

PROGRAMME 6.6 – Propriété P_4

Dans la couche modélisation, la taille de l'image en entrée de la FFT est mémorisée dans la variable auxiliaire `fft_size`. La taille de l'image en sortie de la FFT est récupérée dans la variable `write_length_value` à la lecture du registre `fft_reg::a_write_length` par le `leon_b`. Cette opération est nécessaire car un traitement doit être fait pour récupérer la donnée depuis le pointeur en paramètre de la fonction `read_block`. La valeur calculée est récupérée dans la variable auxiliaire `write_length_value` afin d'être comparée dans l'assertion à `fft_size`. L'observation est faite sur les appels (CALL) de la fonction “`write_block()`” et sur les retours (END) de la fonction “`read_block()`” du `leon_b`. En effet, la valeur lue n'est disponible qu'à la fin de cette fonction.

6.2.1.3 P_5 : Non perte des paquets en entrée

La propriété P_5 permet de vérifier que “*chaque paquet de donnée (image) en entrée dans le IO doit être lu avant l'arrivée d'un autre paquet au niveau du IO*”. Il s'agit alors de vérifier qu'à chaque fois que le module IO émet une requête indiquant la présence d'un nouveau paquet, alors le DMA_a doit finir de lire le paquet avant la génération de la prochaine requête.

```

vunit prop5 {
  // no loss on input packets
  // HDL_DECLs :
  const unsigned long io_module_address = 536870912;
  const unsigned int start_transfer = dma_reg::
    dma_control_start_mask;

  // PROPERTY :
  assert

```

```

always
( io_module.generate_irq_CALL()
  -> next( (dma_a.read_block_END() &&
            dma_a.read_block.p1 == io_module_address)
            before! (io_module.generate_irq_CALL())
          )
);
}

```

PROGRAMME 6.7 – Propriété P_5

Dans cette propriété, l'utilisation du point d'observation "END" permet d'indiquer que l'observation doit être faite à la fin de la lecture de tout le paquet dans l'IO. La trace est échantillonnée sur les appels de la fonction "generate_irq()" de l'IO et sur les retours de la fonction "read_block()" du DMA_a.

6.2.2 Plateforme et assertions raffinées

Nous avons transformé cette plateforme en deux variantes : la première a été obtenue en introduisant le protocole standard Wishbone tandis que la deuxième implémente le protocole standard AMBA-AHB[ARM08]. Nous allons évoquer uniquement la deuxième transformation dans cette section. En revanche, les résultats comporteront une analyse des assertions raffinées pour les deux variantes.

6.2.2.1 La norme AMBA-AHB

La norme AMBA-AHB est la spécification d'une génération d'architectures de bus de haute performance typiquement utilisés pour les liaisons entre le processeur, le contrôleur DMA et les mémoires. Ce bus est conçu pour être utilisé selon le schéma d'interconnexion d'un multiplexeur central⁵. Ainsi, en plus des interfaces des composants maître et esclave, les interfaces de l'arbitre et du décodeur d'adresses sont définies. La Figure 6.6 illustre l'interface d'un maître AHB.

Chaque composant possède deux entrées de synchronisation "HCLK" et "HRESETn". Les ports "HBUSREQx"⁶, "HLOCKx" et "HGRANTx" sont réservés aux maîtres et connectés à des signaux d'arbitrage. Avant de démarrer son transfert, chaque maître doit envoyer une requête via son port HBUSREQ et attendre l'autorisation de l'arbitre du bus (activation de l'entrée HGRANT). La sortie "HLOCK" permet d'indiquer si le maître demande un accès exclusif au bus.

Un maître ayant l'accès au bus commence le transfert en envoyant le signal d'adresse et les signaux de contrôle. Ces signaux fournissent des informations sur la destination,

⁵c'est un bus composé d'un arbitre et d'un décodeur qui contrôlent la transmission et l'orientation des signaux

⁶x indique l'indice du maître

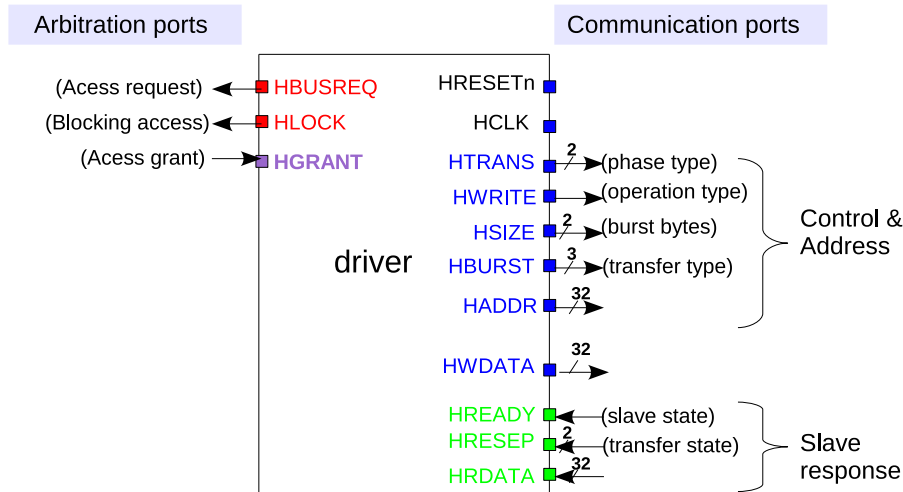


FIGURE 6.6 – Interface d’un maître AHB

le type, la taille du transfert. Deux bus de données sont utilisés : un pour la lecture “HRDATA” et un pour l’écriture “HWDATA”. Chaque transfert est composé d’un cycle de contrôle et d’adresse pendant lequel $HTRANS='2'$ (ou NONSEQ) suivi d’un ou plusieurs cycles de données pendant lesquels $HTRANS='3'$ (ou SEQ). Cette configuration “pipelinée” des transferts est la principale cause de performance du bus AHB.

L’interface des esclaves est symétrique par rapport aux signaux de communication de l’interface maître. Elle contient en plus un port connecté à la sortie du décodeur d’adresses (“HSELx”). Quand elle est à '1', cette entrée indique au composant qu’il est la cible du transfert. Le port de sortie “HREADY” indique au maître la disponibilité de l’esclave. La sortie “HRESP” indique l’état du transfert (échoué, réussi, pas exécuté).

Par conséquent, dans un schéma d’interconnexion multiplexé, le démarrage d’une opération nécessite au préalable la satisfaction d’une pré-condition qui spécifie que le maître est autorisé par le bus et que l’esclave est prêt à recevoir la requête (i.e. HGRANT et HREADY sont à l’état haut). Le démarrage effectif correspond à l’envoi des signaux de contrôle et d’adresse durant la phase d’adresse. La fin d’une opération simple coïncide avec l’envoi des données et de l’indication de réussite du transfert ($HRESP=0$) tout en ayant HREADY à '1'. Pour les transferts en rafale, la taille totale du transfert est déterminée à la fois par le nombre de transferts donné par “HBURST” et par la taille de chaque rafale indiquée dans “HSIZE”.

6.2.2.2 Le raffinement des assertions

Dans la plateforme raffinée, toutes les fonctions de communication qui passent par le bus ont été remplacées par des signaux AHB synchrones. Cependant, nous avons conservé le fonctionnement transactionnel des fonctions de génération d’interruptions (“generate_irq()”) de telle sorte à obtenir, au final, une plateforme hybride.

Des transferts AHB simples ont été implémentés pour réaliser les opérations de configuration des registres. Des transferts en rafale ont été implémentés pour la transmission des données de l'image (paquet). Une horloge globale a été introduite. La fréquence de fonctionnement de cette horloge a été calculée en fonction des latences de fonctionnement des processeurs définies au niveau transactionnel.

Nous avons raffiné les propriétés que nous avons décrites, en utilisant notre outil prototype. La transformation sera détaillée pour la première assertion. Pour les deux autres propriétés, une description des principales étapes sera donnée.

Dans toutes les propriétés raffinées suivantes, les valeurs des signaux sont données en décimal. Nous donnons ici les signaux les plus significatifs utilisés par la suite :

- pour les signaux HTRANS, la valeur 2 correspond à la phase d'adresse ou **NONSEQ** et la valeur 3 à la phase de données ou **SEQ**.
- concernant les signaux HBURST, la valeur 0 indique un transfert simple ou **SINGLE**, une valeur différente indique un transfert en rafale. En particulier, la valeur 7 (ou INC16) correspond à un transfert en rafale à adresses incrémentales composé de 16 transferts simples.
- pour les signaux HSIZE, la valeur 2 indique une données de taille 32 bits (*word*).

P_3 : Configuration du DMA_a

Dans cette assertion, le raffinement concerne uniquement la méthode “**write_block()**” du **Leon_a**. Après l'étape de sélection des fonctions à raffiner, l'utilisateur doit établir les correspondances structurelles entre l'appel de la fonction “**write_block()**” et les signaux de la plateforme raffinée (*c.f.* Figure D.1). Ainsi, comme pour les autres propriétés raffinées, il est impératif d'identifier la signification des termes “début” d'une opération d'écriture simple par le processeur **leon_a** dans la plateforme raffinée. L'outil génère ensuite le modèle de tableau de la Figure 6.7 (rempli par l'utilisateur comme indiqué dans la figure).

Les signaux qui figurent dans le chronogramme correspondent à une observation du début (CALL) d'une opération d'écriture simple selon le protocole AHB. Le chronogramme indique qu'il existe un retard d'un cycle entre l'envoi de l'adresse et des signaux de contrôle, et l'envoi des données. Une contrainte temporelle est alors générée. La transformation temporelle consiste à appliquer, dans l'ordre, les règles de transformation T_6 et T_1 (*c.f.* Programme D.3.1). Au final, la transformation structurelle donne en sortie l'assertion décrite dans le programme 6.8. Le fichier d'instrumentation correspondant est fourni par le programme D.5 en annexe.

```
vunit rtl_prop3 {
// no DMA_a configuration before the end of the current transfer
// HDL_DECLS :
    const unsigned long ctrl_reg = 16777216 +
```

		cycle 0: pre-condition	cycle 1: address phase	cycle 2: data phase
Signals Evolution for method :		write_block	Clock:	clk.POSEDGE
signal	cycle	0	1	2
platform/leon_a_HWRITE			1	
platform/leon_a_HGRANT		1		
platform/leon_a_HTRANS			2	3
platform/leon_a_HBURST			0	
platform/leon_a_HSIZE			2	
platform/bus_a_HREADY		1	1	
platform/leon_a_HWDATA				x
platform/leon_a_HADDR			x	
write_block start condition		leon_a_HTRANS == 2		

FIGURE 6.7 – Chronogramme correspondant à la fonction write_block et à ses paramètres

```

                                dma_reg::a_dma_control;
    const unsigned int start_transfer =
                                dma_reg::dma_control_start_mask;

// HDL_STMTs :
// PROPERTY :
assert
(always
  (prev(leon_a_HGRANT,1,HCLK) && prev(bus_a_HREADY,1,HCLK) &&
    leon_a_HWRITE && leon_a_HTRANS == 2 && leon_a_HBURST == 0 &&
    leon_a_HSIZE==2 && leon_a_HADDR == ctrl_reg
    -> next! (leon_a_HTRANS == 3 && (leon_a_HWDATA==start_transfer)
    -> next (((prev(leon_a_HGRANT,1,HCLK) &&
      prev(bus_a_HREADY,1,HCLK) && leon_a_HWRITE &&
      leon_a_HTRANS == 2 && leon_a_HBURST == 0 &&
      leon_a_HSIZE == 2 && leon_a_HADDR == ctrl_reg
      -> next! (! (leon_a_HTRANS == 3 &&
        leon_a_HWDATA == start_transfer))))
        until!(dma_a.generate_irq_CALL())
    )
  && next_event!(dma_a.generate_irq_CALL())(
    (prev(leon_a_HGRANT,1,HCLK) &&
    prev(bus_a_HREADY,1,HCLK) &&
    leon_a_HWRITE && leon_a_HTRANS == 2 &&
    leon_a_HBURST == 0 && leon_a_HSIZE == 2 &&
    leon_a_HADDR == ctrl_reg
    -> next! (! (leon_a_HTRANS == 3 &&
      leon_a_HWDATA == start_transfer) ) )

```

```

        )
    )
)@ (dma_a.generate_irq_CALL() || HCLK.rising());
}

```

PROGRAMME 6.8 – Propriété P_3 raffinée

Étant donné que la fonction “generate_irq()” est toujours au niveau transactionnel, l’observation de l’assertion hybride résultante se fait sur les fronts montants de l’horloge HCLK et sur les appels de cette fonction transactionnelle “generate_irq()”. La prochaine étape de raffinement permettant d’obtenir l’assertion RTL pourrait intervenir lors de la concrétisation de la fonction generate_irq() par un signal d’interruption.

Dans l’assertion raffinée, la couche modélisation a aussi changé. Les variables globales ont été éliminées. En effet, aucun calcul intermédiaire n’est nécessaire pour récupérer la donnée transmise sur le bus. Celle-ci est directement disponible sur le signal leon_a_HWDATA. Ainsi, dans l’assertion raffinée, la variable data_ptr a été supprimée et value a été remplacée par le signal leon_a_HWDATA. Ces deux modifications ont lieu durant l’étape de transformation des variables globales. Le fichier Excel de transformation est illustré par la Figure 6.8.

Property Variables Transformation			
* If the variable is not used at the RTL level then don't fill any field.			
* If the variable is replaced by an other variable or is equivalent to another input then give this variable/input into the second field and don't fill the vari			
* If the variable is used at the rtl level then give it's new type and it's value			
Global Variables of the Property :	prop1		
Variable Name	replace in RTL with input	RTL Variable Type	RTL Variable Value
data_ptr			
value	leon_a.write_block.p2		

FIGURE 6.8 – fichier Excel de transformation des variables globales de P_3

P_4 : Configuration de la FFT

Le raffinement temporel de cette propriété consiste à intégrer la contrainte temporelle de l’AHB dans l’opération de lecture simple “read_block()” initiée par le leon_b. Pour la fonction d’écriture simple, il n’y aura pas de transformation temporelle puisque uniquement les signaux de contrôle et d’adresses sont concernés et qu’il n’y a pas de délai pour leur transmission.

La fin de lecture simple dans AHB correspond au cycle de données durant lequel le signal leon_b_HTRANS=3. Pendant ce cycle l’esclave envoie les données sur l’entrée

HRDATA du `leon_b` via le signal `leon_b_HRDATA`. En cas du succès de l'opération, le signal `leon_b_HRESP` est égal à 0 et `bus_b_HREADY` est égal à 1.

Le raffinement de cette assertion consiste à appliquer la règle de transformation T_2 sur l'opérateur **before!**. Nous rappelons cette règle :

$$T_2 : (\varphi \text{ and } \psi) \text{ before! } \zeta \xrightarrow{C} (\text{prev}(\varphi, X) \text{ and } \psi) \text{ before! } \zeta$$

$C : \psi \text{ occurs } X \text{ cycles after } \varphi$

Dans cette règle, les variables φ , ψ et ζ correspondent au niveau transactionnel, respectivement aux expressions :

```
- leon_b.read_block.p1 == a_write_length,
- leon_b.read_block_END() && write_length_value == fft_size,
- leon_b.write_block_CALL() && leon_b.write_block.p1 == a_read_addr
```

Selon la Figure D.2 présentée dans l'annexe D, l'adresse est envoyée un cycle avant la fin de l'opération de la lecture (END) et la réception des données par le `leon_b`. X vaut donc 1.

L'assertion obtenue après le remplacement des expressions transactionnelles par les signaux correspondants est donnée dans le programme 6.9. Nous soulignons que pour cette assertion toutes les fonctions de communication observées sont raffinées. Ainsi, l'assertion obtenue est au niveau RTL.

```
vunit rtl_prop4 {
// good FFT configuration
// HDL_DECLs :

    unsigned int fft_size;
    const unsigned long fft_address = 33554432;
    const unsigned long a_read_addr = fft_address +
                                   fft_reg::a_read_addr;
    const unsigned long a_read_length = fft_address +
                                   fft_reg::a_read_length;
    const unsigned long a_write_length = fft_address +
                                   fft_reg::a_write_length;

// HDL_STMTs :
    if(prev(1,leon_b_HWRITE,HCLK) && leon_b_HTRANS==3 &&
    prev(1,leon_b_HBURST,HCLK)==0 &&
    prev(1,leon_b_HSIZE,HCLK) == 2 && bus_b_HREADY &&
    HCLK.rising() &&
    prev(1,leon_b_HADDR,HCLK) == a_read_length)
    {
        fft_size = leon_b_HWDATA;
    }

// PROPERTY :
assert
(always
```

```

    (prev(leon_b_HGRANT,1,HCLK) && prev(bus_b_HREADY,1,HCLK) &&
     leon_b_HWRITE && leon_b_HTRANS==2 && leon_b_HBURST==0 &&
     leon_b_HSIZE==2 && leon_b_HADDR == a_read_addr
     -> next (
        (prev(leon_b_HADDR,1,HCLK) == a_write_length &&
         !leon_b_HWRITE && leon_b_HSIZE == 2 &&
         leon_b_HTRANS == 3 && leon_b_HRESP == 0 &&
         bus_b_HREADY && leon_a_HBURST == 0 &&
         leon_b_HRDATA == fft_size)
         before! (prev(leon_b_HGRANT,1,HCLK) &&
                  prev(bus_b_HREADY,1,HCLK) &&
                  leon_b_HWRITE && leon_b_HTRANS == 2
                  &&
                  leon_b_HBURST == 0 &&
                  leon_b_HSIZE == 2 &&
                  leon_b_HADDR == a_read_addr
                  )
        )
    )
)@(HCLK.rising());
}

```

PROGRAMME 6.9 – Propriété P_4 raffinée

Un exemple de la trace d'exécution est fourni en annexe (D.3.2.1).

 P_5 : Non perte de paquets

L'assertion consiste à s'assurer que le paquet entier de données est lu avant l'arrivée du paquet de données suivant. Pour cette raison, dans l'assertion transactionnelle, l'observation se fait à la fin d'une opération de lecture. Le paramètre (l'adresse) correspond à l'adresse de base du IO.

Au niveau RTL, l'assertion doit conserver ce même traitement (vérifier la lecture du paquet entier). Il est alors nécessaire de couvrir, au niveau RTL, tous les instants entre :

- le début de la lecture du paquet :
 - “dma_a.read_block_CALL()” et
 - “dma_a.read_block.p1 == io_module_address”,
- la fin de la lecture du paquet :
 - “dma_a.read_block_END()” et
 - “dma_a.read_block.p1 == io_module_address+ *taille du paquet*”.

Il faut souligner que seul le premier paramètre de la fonction de lecture est utilisé dans l'assertion. Ainsi, le raffinement des expressions du type “dma_a.read_block_CALL() && dma_a.read_block.p1 == VAR_p1” ne nécessite l'application d'aucune règle de transformation temporelle.

Contrairement aux autres opérations raffinées dans les deux précédentes assertions, la fonction `read_block` du `DMA_a` se traduit par une succession de transferts en rafale afin de permettre la lecture d'une image entière. La taille du paquet à l'entrée du module IO (variable "*taille du paquet*") est fixée dans le fichier "*main*" de la plateforme à l'aide la constante `packet_size`. Elle est configurée à la valeur 16000, ce qui correspond à 4000 mots.

Dans la plateforme raffinée, le transfert de l'image sur le bus est implémenté à l'aide d'une suite de transferts en rafale de type `HBURST='7'`. Chacun de ces transferts commence par un cycle où `HTRANS='2'` pendant lequel la première adresse (A_1) est envoyée suivi de 15 cycles pendant lesquels la donnée (D_i) correspondant à l'adresse précédente ainsi que la nouvelle adresse (A_{i+1}) sont envoyées. Si aucun délai n'est introduit⁷ alors, la dernière adresse (A_{16}) est envoyée au 16^{ième} cycle et la dernière donnée (D_{16}) est envoyée au cycle suivant. À chaque transfert élémentaire, un mot (*word*) est transféré sur le bus des données. Ainsi, selon la configuration considérée, 250 transferts en rafale sont nécessaires pour transférer un paquet de 4000 mots.

Le début d'un transfert en rafale est similaire à celui d'un transfert simple (`HTRANS=2`). La principale différence réside dans la valeur de `HBURST`. La fin d'un transfert en rafale AHB se caractérise, selon la norme, par le passage du signal `HGRANT` à 0 lors du transfert de l'avant dernière adresse, tout en ayant `HREADY` à l'état haut comme le montre la Figure 6.9 cycle T_7 .

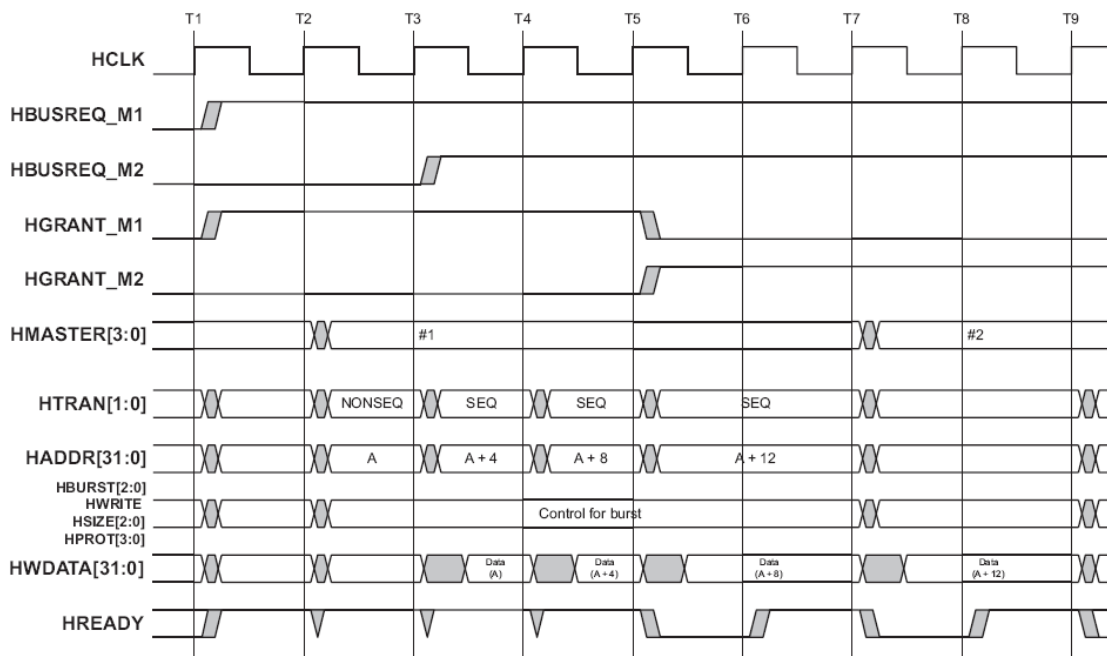


FIGURE 6.9 – La fin d'un transfert en rafale AHB[ARM08]

Cette figure décrit un accès concurrent au bus par deux maîtres M1 et M2 dont chacun

⁷en utilisant le signal `HREADY`

envoie son signal de demande d'accès au bus, respectivement HBUSREQ_M1 et HBUSREQ_M2. Le bus autorise l'accès au premier maître (HGRANT_M1) pour lui permettre d'effectuer un transfert en rafale de taille 4 (composé de quatre transferts élémentaires) aux adresses (A, A+4, A+8 et A+12). La phase HTRANS = NONSEQ correspond au premier cycle du transfert en rafale. Le signal HGRANT_M1 passe à zéro avant le début du dernier transfert d'adresse (T_6). Cette transition marque la fin du transfert en rafale du premier maître. Le transfert contient deux cycles d'attente pendant lesquels HREADY='0'. Ces cycles correspondent à des délais introduits par l'esclave.

Au niveau RTL, s'assurer de la lecture du paquet entier consiste à vérifier toutes les lectures élémentaires ayant lieu entre le début du premier et la fin du dernier transfert en rafale. Dans l'assertion générée (programme 6.10), ce dernier instant est spécifié en utilisant l'expression constituant l'opérande gauche de l'opérateur **before!**. Cette expression est déduite à partir du chronogramme donné par l'utilisateur, illustré par la Figure 6.10.

Signals Evolution for method :		read_block	Clock:	HCLK.POSEDC
signal	cycle	0		
platform/dma_a_HGRANT		0		
platform/dma_a_HTRANS		3		
platform/dma_a_HBURST		7		
platform/dma_a_HSIZE		2		
platform/dma_a_HWRITE		0		
platform/bus_a_HREADY		1		
platform/bus_a_HRESP		0		
platform/dma_a_HADDR		x		
read_block end condition		bus_a_HRESP == 0		

FIGURE 6.10 – Chronogramme correspondant à read_block_END (transfert en rafale)

```

vunit rtl_prop5 {
// no loss on input packets
// HDL_DECLS :
    const unsigned long io_module_address = 536870912;
    const unsigned int start_transfer =
        dma_reg::dma_control_start_mask;
    const unsigned int packet_size = 16000;
// HDL_STMTs :
// PROPERTY :
assert
(always
    (io_module.generate_irq_CALL()
    -> next(
        (!dma_a_HGRANT && !dma_a_HWRITE &&

```

```

        dma_a_HTRANS ==3 && dma_a_HBURST == 7 &&
        dma_a_HSIZE == 2 &&
        bus_a_HRESP == 0 &&
        dma_a_HADDR ==
            (io_module_address + (packet_size - 4))
            before! (io_module.generate_irq_CALL() )
    )
)
)@(HCLK.rising() || io_module.generate_irq_CALL());
}

```

PROGRAMME 6.10 – Propriété P_5 raffinée

Une modification a été introduite par rapport à l’assertion obtenue en sortie de l’outil de raffinement : la valeur de l’adresse de la lecture qui pointe vers la première adresse de lecture dans le bloc IO a été incrémentée de la taille du paquet entier moins quatre afin de pointer vers l’adresse du dernier mot du paquet de données. Cette information n’est pas automatisable.

L’assertion générée par l’outil permet uniquement de détecter la fin de la lecture d’un bloc de donnée. Pour que l’assertion raffinée soit encore plus fidèle à l’assertion de départ, il faudrait prendre en considération la lecture du paquet de données en entier (entre “dma_a_HADDR= io_module_address” et “dma_a_HADDR = io_module_address+packet_size”). L’ajout de ce traitement se ferait principalement dans la couche de modélisation de la propriété de la manière suivante :

```

// HDL_STMTs
if( dma_a_HADR == io_module_address &&
    !dma_a_HWRITE && dma_a_HTRANS==2 &&
    dma_a_HBURST==7 && dma_a_HSIZE==2) {
    read_started = true;
}

if( dma_a_HADR == io_module_address+packet_size &&
    !dma_a_HWRITE && dma_a_HTRANS==3 &&
    dma_a_HBURST==7 && dma_a_HSIZE==2 ) {
    read_started = false;
}

// PROPERTY :
(always
    (io_module.generate_irq_CALL()
        -> next(read_started && ...

```

Il s’agit ici, d’une limitation de l’outil de raffinement qui, dans sa version actuelle ne fournit pas de support pour la transformation des transferts en rafale. Nous reviendrons sur ce point dans la section des travaux futurs (*c.f.* section 7.2).

6.2.2.3 Exemple de raffinement en utilisant les SEREs

Nous avons vu dans le chapitre 4 à la section 4.3.2, que les expressions mises en jeu dans les règles T_2 et T_6 se raffinaient naturellement en SERE. La mise en oeuvre des SEREs dans ISIS permet cette alternative. Nous proposons de prendre le cas de la propriété P_4 comme exemple. La règle de raffinement applicable sur cette propriété est T_2 . Nous rappelons que la transformation intermédiaire de la partie gauche de ce patron de formule en une SERE se fait de la manière suivante (6.2.2.3) :

T_2 -SERE : $(\varphi \text{ and } \psi) \text{ **before!** } \zeta \xrightarrow{C}$
 $\{(\neg\zeta)[*]; \neg\zeta \wedge \varphi; (\neg\zeta \wedge \neg\psi)[*(X-1)]; \neg\zeta \wedge \psi; [*]\}$
 $C : \psi \text{ occurs } X \text{ cycles after } \varphi$

La règle de transformation s'applique sur le nœud “**before!**” de la propriété. Les formules φ , ψ et ζ correspondent respectivement aux expressions :

```
- leon_b.read_block.p1 == a_write_length,
- (leon_b.read_block_END() && leon_b.read_block.p2 == fft_size,
- (leon_b.write_block_CALL() && leon_b.write_block.p1 == a_read_addr).
```

Nous rappelons ici que X vaut 1. Ainsi, la SERE sera en fait :

$\{(\neg\zeta)[*]; \neg\zeta \wedge \varphi; \neg\zeta \wedge \psi; [*]\}$

À la fin du processus de raffinement temporel, la formule obtenue est :

```
always(leon_b.write_block_CALL() &&
  leon_b.write_block.p1 == a_read_addr
  -> next (
    {!(leon_b.write_block_CALL() &&
      leon_b.write_block.p1 == a_read_addr)[*];
      (!(leon_b.write_block_CALL() &&
        leon_b.write_block.p1 == a_read_addr) &&
      (leon_b.read_block_END() &&
        leon_b.read_block.p1 == a_write_length) );
      (!(leon_b.write_block_CALL() &&
        leon_b.write_block.p1 == a_read_addr) &&
      (leon_b.read_block.p2 == fft_size));
      [*]
    })
  );
```

PROGRAMME 6.11 – Raffinement temporel en SERE de la propriété P_4

Le raffinement structurel consiste à remplacer les expressions de communication transactionnelles par les signaux correspondants dans la plateforme raffinée. Ceci revient à remplacer dans l'assertion 6.11, les variables φ , ψ et ζ par :

```
-  $\varphi$  : leon_b_HADDR == a_write_length.
```

- ψ : `leon_b_HTRANS == 3 && leon_b_HRESP == 0 && bus_b_HREADY && !leon_b_HWRITE && leon_b_HSIZE == 2 && leon_b_HBURST == 0 && leon_b_HRDATA == fft_size.`
- ζ : `prev(leon_b_HGRANT,1,HCLK) && prev(bus_b_HREADY,1,HCLK) && leon_b_HWRITE && leon_b_HTRANS == 2 && leon_b_HBURST == 0 && leon_b_HSIZE == 2 && leon_b_HADDR == a_read_addr.`

En appliquant ces changements, l'assertion obtenue est :

```
vunit rtl_prop4 {
// good FFT configuration
// HDL_DECLs :

    unsigned int fft_size;
    const unsigned long fft_address = 33554432;
    const unsigned long a_read_addr = fft_address +
                                fft_reg::a_read_addr;
    const unsigned long a_read_length = fft_address +
                                fft_reg::a_read_length;
    const unsigned long a_write_length = fft_address +
                                fft_reg::a_write_length;

// HDL_STMTs :
    if(prev(1,leon_b_HWRITE,HCLK) && leon_b_HTRANS==3 &&
        prev(1,leon_b_HBURST,HCLK)==0 &&
        prev(1,leon_b_HSIZE,HCLK) == 2 && bus_b_HREADY &&
        HCLK.rising() &&
        prev(1,leon_b_HADDR,HCLK) == a_read_length)
    {
        fft_size = leon_b_HWDATA;
    }

// PROPERTY :
assert
(always
    (prev(leon_b_HGRANT,1,HCLK) && prev(bus_b_HREADY,1,HCLK) &&
    leon_b_HWRITE && leon_b_HTRANS==2 && leon_b_HBURST==0 &&
    leon_b_HSIZE==2 && leon_b_HADDR == a_read_addr
    -> next (
        {!(prev(leon_b_HGRANT,1,HCLK) &&
            prev(bus_b_HREADY,1,HCLK) &&
            leon_b_HWRITE && leon_b_HTRANS==2 &&
            leon_b_HBURST==0 && leon_b_HSIZE==2 &&
            leon_b_HADDR == a_read_addr)[*];
            (!(prev(leon_b_HGRANT,1,HCLK) &&
                prev(bus_b_HREADY,1,HCLK) &&
                leon_b_HWRITE && leon_b_HTRANS==2 &&
                leon_b_HBURST==0 && leon_b_HSIZE==2 &&
                leon_b_HADDR == a_read_addr) &&
                leon_b_HADDR== a_write_length);
```

```

        (!(prev(leon_b_HGRANT,1,HCLK) &&
prev(bus_b_HREADY,1,HCLK) &&
        leon_b_HWRITE && leon_b_HTRANS==2 &&
        leon_b_HBURST==0 && leon_b_HSIZE==2 &&
        leon_b_HADDR == a_read_addr) &&
        (leon_b_HTRANS == 3 && leon_b_HRESP == 0 &&
        bus_b_HREADY && !leon_b_HWRITE &&
        leon_b_HSIZE == 2 && leon_b_HBURST == 0 &&
        leon_b_HRDATA == fft_size) );
    [*]})
)
)@(HCLK.rising());
}

```

PROGRAMME 6.12 – Propriété P_4 raffinée en SERE

Cette assertion signifie que si le `leon_b` finit de programmer la FFT (i.e. écrit la taille du transfert dans le registre `a_read_addr`) alors il ne faut pas que cet événement se reproduise avant d’avoir une lecture dans le registre `a_write_length` telle que la donnée lue soit égale à `fft_size`.

Un extrait de la trace d’exécution est fourni en annexe (D.3.2.2).

6.3 Résultats et analyses

6.3.1 Complexification des assertions raffinées

Cette analyse vise à donner une idée plus claire sur la complexification des assertions suite à l’intégration d’un protocole de communication dans la plateforme raffinée. Elle est basée sur les données des propriétés traitées dans ce chapitre. Les tableaux récapitulatifs 6.11 et 6.12 contiennent respectivement les caractéristiques de chaque propriété d’origine (au niveau transactionnel) et raffinée (hybride ou au niveau RTL) traitée dans le cadre de la plateforme de décodage vidéo “MJPEG”. Pour chaque propriété, nous fournissons le nombre d’opérateurs PSL, le nombre d’expressions booléennes élémentaires ainsi que le nombre de règles de transformation appliquées lors de l’étape de raffinement temporel. Étant donné que le protocole de communication implémenté dans la plateforme raffinée est le protocole Wishbone, aucune transformation temporelle n’a été introduite sur les assertions d’origine. Le nombre d’opérateurs PSL temporels est alors invariant. La transformation est purement structurelle. Nous remarquons alors que le nombre d’expressions booléennes a augmenté. La variation du nombre total d’opérateurs PSL s’explique principalement par l’introduction des fonctions PSL de la couche booléenne (**prev**, **rose**). Bien qu’il ne s’agisse pas d’opérateurs mais de fonctions PSL, nous les prenons en compte dans le nombre total.

propriété TLM	Nombre d'opérateurs PSL		Nombre d'expressions
	temporels	total	booléennes
P1	3	4	6
P2	2	3	4

FIGURE 6.11 – M-JPEG : caractéristiques des propriétés TLM P_1 , et P_2

propriété raffinée (RTL/hybride)	Nombre d'opérateurs PSL		Nombre d'expressions booléennes	Nombre de règles de transformation
	temporels	total		
P1	3	6	9	0
P2	2	4	5	0

FIGURE 6.12 – M-JPEG : complexification des propriétés raffinées P_1 , et P_2

Les tableaux 6.13 et 6.14 concernent le raffinement des assertions de la plateforme de traitement d'images. Les résultats du raffinement sont reportés pour les deux versions de la plateforme raffinée : en utilisant le protocole Wishbone et le protocole AMBA-AHB. Les résultats montrent que les assertions générées relativement au protocole AHB sont plus complexes que leurs équivalentes générées relativement au protocole Wishbone. Ceci est notamment vrai pour la propriété P_3 où deux règles de transformation ont été appliquées lors du raffinement temporel. Hormis le fait que le protocole AHB est plus complexe temporellement, le nombre de signaux nécessaires pour implémenter ce dernier est supérieur à celui du protocole Wishbone.

propriété TLM	Nombre d'opérateurs PSL		Nombre d'expressions booléennes
	temporels	total	
P3	4	5	9
P4	4	5	9
P5	4	5	6

FIGURE 6.13 – Traitement d'images : caractéristiques des propriétés TLM P_3 , P_4 et P_5

Il faut noter, en revanche, que le raffinement de la propriété P_4 en SERE en utilisant la règle de transformation T_2 -SERE a donné lieu à une assertion nettement plus complexe avec 8 opérateurs temporels contre 4 pour l'assertion FL.

Le raffinement des mêmes assertions de la plateforme de traitement d'image selon deux protocoles de communication de bas niveau est particulièrement intéressant car il permet de mesurer l'effet de la variation du protocole de communication sur la complexité des assertions raffinées.

6.3.2 Analyse des performances

Dans cette partie, nous allons reporter les temps de simulation des plateforme transactionnelles et raffinées non instrumentées et instrumentées par chacune des propriétés traitées. Notons que les mesures qui seront données ont été faites sur un processeur Intel Core2 Duo sous la distribution Debian de Linux.

	propriété raffinée (RTL/hybride)	Nombre d'opérateurs PSL		Nombre d'expressions booléennes	Nombre de règles de transformation
		temporels	total		
Wishbone	P3	4	7	14	0
	P4	4	7	18	0
	P5	4	5	10	0
AHB	P3	8	18	34	2
	P4	4	10	23	1
	P4-SERE	8	17	37	1
	P5	4	5	12	0

FIGURE 6.14 – Traitement d'images : complexification des propriétés raffinées P_3 , P_4 et P_5

6.3.2.1 Plateforme de décodage vidéo (MJPEG)

Les Figures 6.15 et 6.16 montrent les résultats de simulation pour la plateforme MJPEG transactionnelle pour le décodage d'une vidéo donnée. Les Figures 6.17 et 6.18 montrent les résultats de simulation pour la plateforme raffinée au niveau RTL pour le décodage de la même vidéo.

Dans la Figure 6.15, chaque valeur de l'axe des abscisses correspond à un temps de décodage vidéo donné en secondes. Ce temps peut être spécifié au début de chaque simulation. Les valeurs de l'axe des ordonnées correspondent à la moyenne des mesures des temps d'exécution (CPU) de la plateforme transactionnelle. Les trois courbes identifiées dans la légende de la figure décrivent, respectivement, les résultats de simulation de la plateforme non instrumentée, instrumentée avec la propriété P_1 et instrumentée avec la propriété P_2 . Dans la Figure 6.17, les mêmes temps de décodages ont été repris pour effectuer les mesures sur la plateforme du niveau RTL.

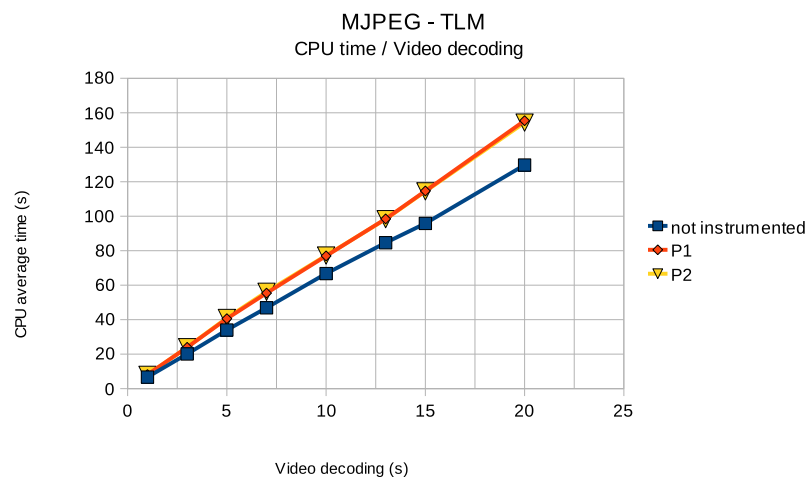


FIGURE 6.15 – Plateforme de décodage vidéo au niveau transactionnel : temps CPU pour les propriétés P_1 et P_2

Dans les tableaux des Figures 6.16 et 6.18, le nombre d'activations des moniteurs

des deux propriétés ainsi que la surcharge provoquée par l'ajout de l'infrastructure de vérification sont précisés pour les deux niveaux d'abstraction de la plateforme.

Au niveau transactionnel, le nombre d'activations des moniteurs correspond au nombre d'instances des actions de communication appartenant à la liste de sensibilité du moniteur survenues au cours de la simulation. Étant donné que les moniteurs des deux propriétés sont actifs sur les mêmes fonctions, nous obtenons des nombres d'activations égaux pour le même nombre de décodage vidéo.

Nous notons que les surcharges provoquées par les propriétés surveillées sont limitées et quasiment équivalentes. En effet, le calcul effectué par les moniteurs est moindre par rapport à celui nécessaire pour les activités de décodage et d'affichage de la vidéo.

Decodage video (sec)	Evaluations de la propriété		% Excédent temps CPU	
	P1	P2	P1	P2
1	8668358	8668358	21,6	19,9
3	25782928	25782928	19,0	19,4
5	43357712	43357712	19,4	20,6
7	59296080	59296080	18,1	19,8
10	81914950	81914950	15,3	15,7
13	103565962	103565962	16,3	16,2
15	120646726	120646726	19,6	19,4
20	162926588	162926588	19,9	19,0

FIGURE 6.16 – Plateforme de décodage vidéo au niveau transactionnel : nombre d'activations des moniteurs de P_1 et P_2

Au niveau RTL, la simulation de la plateforme non instrumentée est en moyenne quinze fois plus lente que son équivalente au niveau transactionnel. Nous retrouvons un facteur assez classique pour le passage des simulations TLM aux simulations RTL ou hybride. Nous notons par contre que les écarts entre les trois courbes se sont estompés car les moniteurs restent, eux d'une efficacité comparable au contexte TLM.

Au niveau RTL, le nombre d'activations correspond au nombre de fronts de l'horloge `clk` pour une simulation donnée.

6.3.2.2 Plateforme de compression d'images

Plusieurs simulations ont été faites sur les différentes descriptions de cette plateforme : transactionnelle, hybride implémentant le protocole Wishbone et hybride implémentant le protocole AHB. Dans cette section, nous réunissons les temps CPU pour certaines des simulations effectuées. Dans les résultats relatés, chaque description a été exécutée plusieurs fois, suivant des simulations de durées croissantes. La durée de la simulation est proportionnelle au nombre d'images traitées. Ce nombre est mentionné pour chaque simulation. Chaque simulation a été effectuée d'abord sans aucune instrumentation par

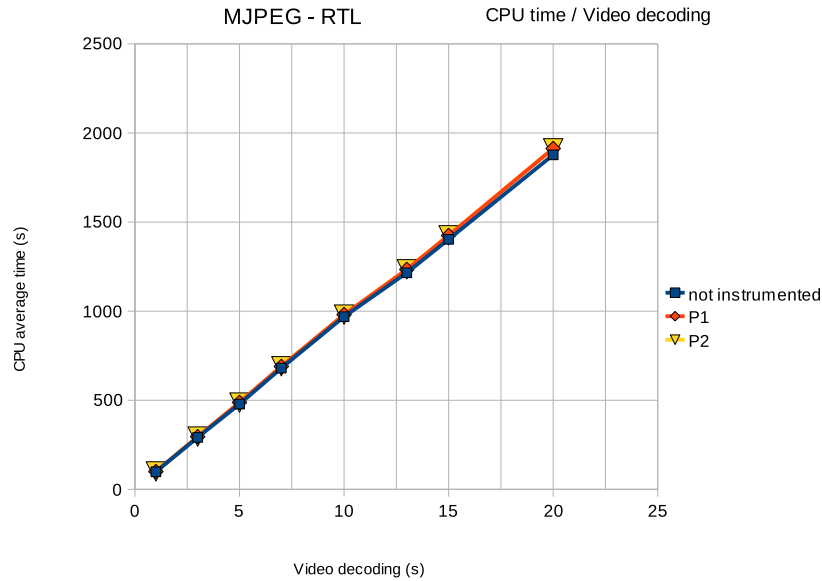


FIGURE 6.17 – Plateforme de décodage vidéo au niveau RTL : temps CPU pour les propriétés P_1 et P_2 raffinées

Décodage vidéo (sec)	Évaluations de la propriété		% Excédent temps CPU	
	P1	P2	P1	P2
1	99999999	99999999	0,9	1,8
3	299999999	299999999	1,3	1,7
5	499999999	499999999	1,9	1,7
7	699999999	699999999	1,4	1,5
10	999999999	999999999	1,2	1,2
13	1299999999	1299999999	1,5	1,6
15	1499999999	1499999999	1,5	1,6
20	1999999999	1999999999	1,9	1,9

FIGURE 6.18 – Plateforme de décodage vidéo au niveau RTL : nombre d'activations des moniteurs de P_1 et P_2 raffinées

ISIS, ensuite avec la surveillance d'une propriété à la fois. Le temps CPU donné pour chaque nombre d'images traitées est la moyenne de cinq simulations identiques.

Le nombre d'images considérées dans les descriptions raffinées a été généralement divisé par dix en raison de la lenteur de la simulation au niveau RTL. Selon les expérimentations, la simulation de la plateforme hybride implémentant le protocole Wishbone est environ huit fois moins rapide que celle effectuée au niveau transactionnel. La simulation de la plateforme hybride implémentant le protocole AHB est huit fois plus lente que celle implémentant le protocole Wishbone.

Le graphe de la Figure 6.19 réunit les résultats de sept simulations de la plateforme transactionnelle non instrumentée et instrumentée avec chacune des trois propriétés considérées. Dans ces mesures, le nombre d'images traitées varie entre 100 et 3000 images sur l'axe des abscisses. La figure montre que l'augmentation du nombre d'images traitées engendre une croissance linéaire du temps CPU aussi bien pour la plateforme non in-

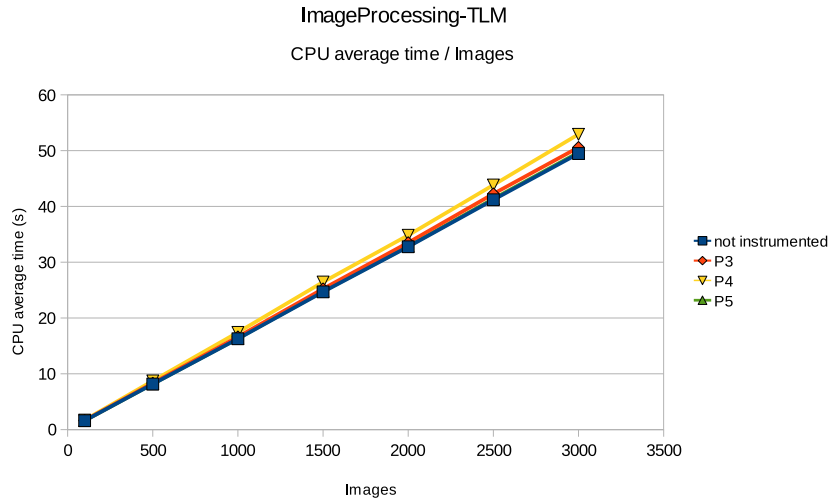


FIGURE 6.19 – Plateforme de traitement d’images au niveau transactionnel : temps CPU pour les propriétés P_3 , P_4 et P_5

strumentée qu’instrumentée. La figure montre aussi que l’excédent de temps CPU dû au fonctionnement des moniteurs est variable selon la propriété. Seule la propriété P_4 induit une augmentation de temps significative.

Nombre d’images	Évaluations de la propriété			% Excédent temps CPU		
	P3	P4	P5	P3	P4	P5
100	1100	77149	200	2,07	7,09	0,28
500	5500	390749	1000	1,92	6,91	0,29
1000	11000	782749	2000	2,11	7,02	0,38
1500	16500	1174749	3000	2,22	7,16	0,47
2000	22000	1566749	4000	2,25	6,29	0,26
2500	27500	1958749	5000	2,56	6,41	0,39
3000	33000	2350749	6000	2,22	6,96	0,37

FIGURE 6.20 – Plateforme de traitement d’images au niveau transactionnel : nombre d’activations des moniteurs de P_3 , P_4 et P_5

Le tableau de la Figure 6.20 illustre le nombre d’activations des moniteurs de chaque propriété pour chaque nombre d’images et précise l’excédent de temps CPU dû au fonctionnement des moniteurs. Les résultats obtenus montrent une relation entre le nombre d’activations du moniteur et l’excédent de temps CPU de chaque assertion. En effet, le nombre d’activations du moniteur de P_4 est nettement plus élevé que celui des deux autres. Cependant, cette surcharge du temps CPU reste négligeable par rapport au temps CPU de la plateforme non instrumentée (ne dépassant pas 8% pour P_4).

Les résultats obtenus concernant les plateformes raffinées sont relatifs à neuf mesures faites pour un nombre d’images allant de 10 à 300 images.

Les Figures 6.21 et 6.22 sont associées à la description hybride de la plateforme raffinée en utilisant le protocole de communication Wishbone. Les propriétés raffinées à l’origine de ces résultats sont données en annexe (*c.f.* section D.2). La surcharge d’exécution est ici plus nette pour les propriétés P_3 et P_5 .

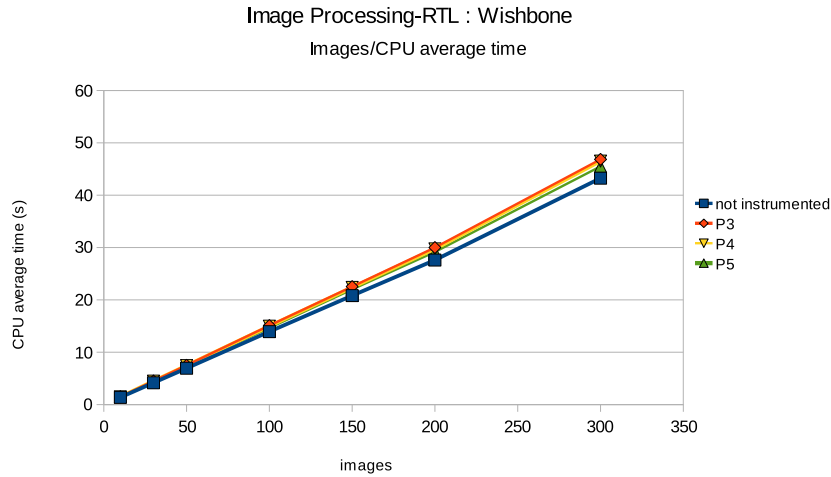


FIGURE 6.21 – Plateforme de traitement d’images hybride implémentant le protocole Wishbone : temps CPU pour les propriétés P_3 , P_4 et P_5 raffinées

Nombre d’images	Évaluations de la propriété			% Excédent temps CPU		
	P3	P4	P5	P3	P4	P5
10	583344	583334	583344	8,09	7,65	5,77
15	875015	875000	875015	8,01	7,61	5,04
20	1166687	1166667	1166687	8,15	7,55	5,52
30	1750030	1750000	1750030	8,36	7,54	4,93
50	2916717	2916667	2916717	8,27	7,39	5,19
100	5833434	5833334	5833434	8,32	7,46	5,09
150	8750150	8750000	8750150	8,44	7,54	5,72
200	11666867	11666667	11666867	8,66	7,75	5,60
300	17500300	17500000	17500300	8,85	7,80	5,22

FIGURE 6.22 – Plateforme de traitement d’images hybride implémentant le protocole Wishbone : nombre d’activations des moniteurs de P_3 , P_4 et P_5 raffinées

Contrairement aux deux autres propriétés, le moniteur de l’assertion P_4 est actif uniquement sur l’observation du front montant de l’horloge `clk`. Le nombre d’activations de ce moniteur correspond par conséquent au nombre de coups d’horloge pour chaque simulation. La différence du nombre d’activations pour les autres propriétés correspond au nombre d’appels de la fonction “`generate_irq()`”.

L’excédent de temps CPU le moins important est celui de la propriété P_5 . À nombre d’activations égal, le moniteur de cette assertion provoque par exemple une surcharge de 5.09% pour 100 images traitées contre 8.32% pour l’assertion P_3 . Cette différence se justifie par le fait que la complexité du moniteur de l’assertion P_5 est restée quasiment inchangée entre le niveau transactionnel et hybride (RTL-TLM). Les tableaux des figures 6.13 et 6.14 montrent que seul le nombre d’expressions booléennes a augmenté pour passer de 6 à 10.

Le tableau 6.14 montre aussi que les moniteurs les plus complexes sont ceux des assertions P_4 et P_3 . Cependant, au niveau du tableau 6.22, l’excédent de temps CPU dû au second moniteur est légèrement plus élevé. Pour 100 images, il est en moyenne

de 8.32% pour P_3 contre 7.46% pour P_4 . La raison derrière cette légère différence réside vraisemblablement dans le nombre d'activations de chaque moniteur. Pour 100 images, le moniteur de P_3 a été activé 100 fois de plus que celui de l'assertion P_4 .

Globalement, la surcharge du temps d'exécution est négligeable (ne dépassant pas 9%).

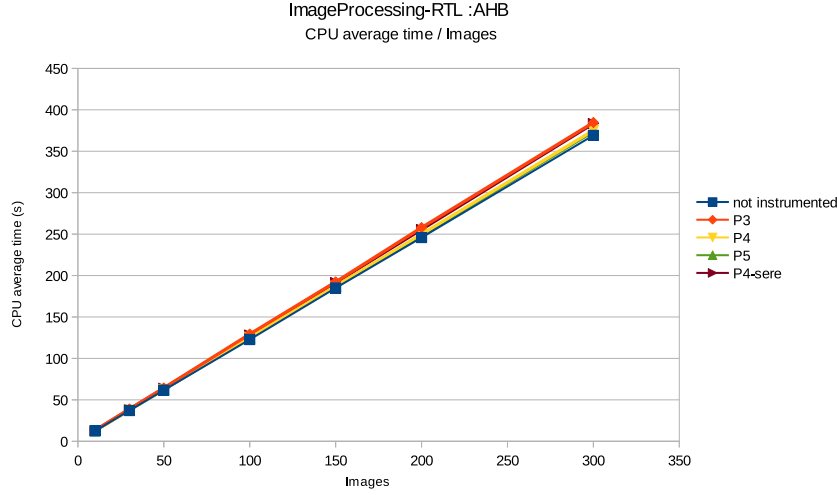


FIGURE 6.23 – Plateforme de traitement d'images hybride implémentant le protocole AHB : temps CPU pour les propriétés P_3 , P_4 et P_5 raffinées

Les Figures 6.23 et 6.24 exposent les résultats obtenus en effectuant les mêmes mesures sur la plateforme raffinée en utilisant le protocole de communication AMBA-AHB. La propriété P_4 est représentée dans ses deux versions raffinées : en FL (P_4) et en SERE(P_4 -SERE).

Dans le tableau de la Figure 6.24, on remarque qu'à nombre d'activations égal, l'excédent de temps CPU relatif à la propriété P_3 est plus important que celui de la propriété P_5 . Pour 100 images, par exemple, il est de 5.36% contre 1.66%. Cette variation est due au fait que le moniteur de P_3 est plus complexe que celui de P_5 (tableau 6.14). En effet, le moniteur de P_3 est composé de 18 opérateurs PSL dont 8 sont temporels. C'est le double du nombre d'opérateurs temporels du moniteur de P_5 .

Les résultats des deux versions de la propriété P_4 raffinées sont différents. Le moniteur de la version SERE étant plus complexe (le double d'opérateurs temporels), l'excédent de temps CPU est plus élevé. Pour 100 images, l'excédent de temps CPU de la version SERE est de 3.87% contre 2.18% pour la version FL.

Les moniteurs des deux assertions P_3 et P_4 -SERE ont le même nombre d'opérateurs temporels. Cependant, la surcharge du temps d'exécution due au premier moniteur est plus importante. Pour 100 activations supplémentaires, la différence de surcharge est de 1.5%.

En conclusion, la surcharge provoquée par l'inclusion de l'infrastructure d'observation dépend à la fois de la complexité structurelle et temporelle du moniteur et du nombre

d'activations de ce dernier.

Dans le cas du protocole AHB, l'excédent de temps CPU dû à l'instrumentation est négligeable malgré l'augmentation de la complexité des moniteurs. Le plus important excédent (celui de P_3) est de $\approx 6\%$ pour AHB contre $\approx 9\%$ pour Wishbone. Le "surcoût" dû à l'instrumentation a diminué car la complexification des moniteurs est négligeable par rapport à la complexification de la plateforme raffinée.

Nombre d'images	Évaluations de la propriété				% Excédent temps CPU			
	P3	P4	P4-SERE	P5	P3	P4	P4-SERE	P5
10	583344	583334	583334	583344	5,32	2,07	3,34	1,78
15	875015	875000	875000	875015	4,83	2,17	3,88	1,04
20	1166687	1166667	1166667	1166687	4,37	1,87	3,67	1,53
30	1750030	1750000	1750000	1750030	5,12	1,53	3,41	1,72
50	2916717	2916667	2916667	2916717	4,25	1,57	3,30	1,23
100	5833434	5833334	5833334	5833434	5,36	2,18	3,87	1,66
150	8750150	8750000	8750000	8750150	4,04	1,25	3,61	1,07
200	11666867	11666667	11666667	11666867	4,89	1,61	4,02	0,99
300	17500300	17500000	17500000	17500300	4,20	1,63	3,33	1,25

FIGURE 6.24 – Plateforme de traitement d'images hybride implémentant le protocole AHB : nombre d'activations des moniteurs de P_3 , P_4 et P_5 raffinées

6.3.3 Synthèse

Dans toutes les propriétés traitées, l'assertion raffinée est souvent plus complexe que l'assertion d'origine. En se référant aux tableaux donnés, nous pouvons déduire que la complexité de l'assertion raffinée dépend à la fois du nombre de règles de transformation appliquées lors de l'étape de raffinement temporel et du protocole de communication sous-jacent. L'utilisation de l'outil de transformation permet d'abstraire la difficulté du raffinement des assertions PSL grâce à l'automatisation de l'intégration des contraintes temporelles et des correspondances structurelles.

Les diverses mesures effectuées sur les cas d'études présentés ont permis de démontrer que malgré la complexité des assertions raffinées, les excédents de temps CPU sont relativement bas. Les résultats reportés concernent une propriété à la fois afin d'isoler les effets de chaque moniteur. Cependant, il est possible d'instrumenter la plateforme en utilisant toutes les assertions à la fois. Dans ce cas, l'excédent de temps CPU ne correspond généralement pas au cumul des excédents de temps CPU distincts, étant donné que plusieurs moniteurs peuvent partager l'observation des mêmes fonctions de communication.

6.4 Bilan

Nous avons montré, dans ce chapitre, quelques résultats d'application de notre méthode de raffinement. Les différentes expérimentations effectuées sur les cas d'études présen-

tés ont permis de valider la méthode de transformation proposée et de démontrer son intérêt.

Les assertions en sortie de l'outil de raffinement ont été employées selon une technique de vérification dynamique (simulation). Nous avons fait appel à la nouvelle version de l'outil ISIS pour la génération des moniteurs de surveillance et pour l'instrumentation du modèle à surveiller. Il est néanmoins nécessaire de souligner que les assertions générées pourraient également être utilisées en vérification formelle (*Model Checking*) ou en émulation.

Conclusion et perspectives

L'utilisation des plateformes virtuelles au niveau transactionnel, et l'utilisation de nouvelles méthodologies et de langages de modélisation de haut niveau tels que SystemC TLM, nécessite l'intégration de techniques de vérification dès les premières étapes de conception, et le raffinement conjoint des modèles et des assertions.

7.1 Contributions

La thèse présentée dans ce manuscrit est une contribution à la construction d'un flot de vérification conjoint au flot de conception des circuits intégrés, à travers l'automatisation du raffinement des assertions. L'ensemble d'assertions spécifiées au niveau transactionnel est une spécification formelle par rapport à laquelle est effectuée la vérification du modèle de la plateforme. Cette démarche permet de renforcer la confiance dans ce modèle qui constitue la base pour la construction des modèles des niveaux d'abstraction inférieurs.

La génération automatique des assertions raffinées permet de vérifier les modèles raffinés par rapport à la même spécification formelle de départ. Pour cette raison, il est primordial de s'assurer de la correction du processus de raffinement.

Raffinement des assertions

Notre solution est composée de deux étapes : une étape de transformation temporelle et une étape de transformation structurelle. La transformation temporelle est basée sur un ensemble de règles de raffinement effectuant une transformation formelle d'une formule PSL en une autre formule PSL en fonction d'une contrainte temporelle liée au protocole implémenté. La transformation structurelle consiste à remplacer les expressions transactionnelles par les signaux du protocole.

De même que pour le processus de raffinement des modèles, le raffinement des assertions n'est pas entièrement automatisable. Il nécessite que les informations temporelles et

structurelles nécessaires au raffinement soient fournies par l'utilisateur. L'outil réalise la saisie de ces informations de façon ergonomique, puis procède automatiquement à la transformation structurelle et temporelle de l'assertion. Il permet la génération d'assertions au niveau RTL mais aussi d'assertions hybrides.

Nos travaux ont été réalisés dans le contexte des approches dynamiques, qui permettent d'éviter le problème d'explosion combinatoire. Nous nous sommes intéressées en particulier à la simulation. Les expérimentations ont été faites en conjonction avec l'outil ISIS de génération automatique des moniteurs de surveillance. Une extension de cet outil a été nécessaire.

Extension d'ISIS

L'extension de l'outil ISIS a porté sur deux axes :

1. L'ajout du support nécessaire pour les assertions hybrides.
2. L'implémentation des moniteurs SERE.

Le premier point se concrétise par l'implémentation des fonctions PSL "rose" et "fell" permettant de prendre en compte les fronts montants ou descendants de signaux, et par l'ajout de la possibilité de consulter la valeur antérieure (fonction PSL "prev") d'un signal.

D'autre part, les principaux opérateurs SERE ont été implémentés afin de permettre d'utiliser les règles de raffinement donnant lieu à des séquences. Actuellement deux règles de transformation sont concernées, mais cette alternative pourrait être utilisée pour d'autres règles de transformation puisque la sémantique des SERE s'adapte parfaitement au contexte RTL.

Bilan

L'outil prototype de raffinement développé fait le lien entre les deux outils ISIS et Horus développés dans l'équipe. L'ensemble de ces outils peut être considéré comme une infrastructure de vérification dynamique couvrant en grande partie le flot de conception. La méthode de raffinement proposée permet d'obtenir des assertions susceptibles d'être aussi utilisées en vérification formelle ou en émulation. À notre connaissance, il n'existe pas encore une telle solution automatisée de raffinement des assertions basée sur des transformations formelles.

Les résultats des expérimentations donnés dans le précédent chapitre ont permis de prouver l'intérêt et l'efficacité de la solution proposée.

7.2 Travaux futurs

Les résultats obtenus sont prometteurs et prouvent l'intérêt de la solution proposée. Il reste néanmoins quelques améliorations à faire avant d'aboutir à une solution plus complète.

7.2.1 Travaux à court terme : génération des chronogrammes

Concernant l'outil de raffinement, la spécification des chronogrammes des fonctions de communication pourrait être améliorée. Une solution est en cours de développement afin de permettre la génération automatique des chronogrammes pour certains protocoles de communication prédéfinis (standard, comme AHB ou Wishbone, ou protocole propriétaire). Pour chaque fonction raffinée, l'utilisateur pourra spécifier le type de l'opération (lecture ou écriture) et le type de transfert au niveau RTL (simple, en rafale, pipeline). En fonction de ces données, l'outil pourra générer le squelette du chronogramme correspondant. Les chronogrammes correspondant aux protocoles standards sont disponibles à partir d'une bibliothèque de fichiers Excel pré-remplis. L'utilisateur devra effectuer la correspondance entre les signaux standards de ces chronogrammes et les signaux effectifs de sa plateforme.

Une amélioration plus sophistiquée pourra être apportée en implémentant les FSM relatives aux protocoles standards tels que décrits dans [KBS11] [SO14], afin de générer les chronogrammes dynamiquement à partir de ces FSMs.

L'apport de l'automatisation de la génération de chronogrammes est double, d'une part elle permettra d'alléger l'intervention de l'utilisateur et d'autre part, elle facilitera la prise en considération des transferts les plus complexes. L'outil pourra par exemple déduire la durée totale d'un transfert en rafale AHB en fonction des valeurs des signaux de contrôle HBURST et HREADY.

7.2.2 Perspectives

Les règles de transformation proposées permettent de prendre en considération un ensemble de contraintes temporelles concernant les transferts simples. Il n'existe pas actuellement de support pour les transferts plus complexes, du type rafale ou pipelinés. Une évolution de cet ensemble est alors nécessaire. Comme nous l'avons constaté avec l'exemple de l'assertion P_5 raffinée (*c.f.* section 6.2.2.2), une utilisation de la couche de modélisation pourrait résoudre certaines de ces limitations. Ainsi, pour vérifier un transferts en rafale, il faudrait pouvoir itérer sur tous les transferts élémentaires pour reconstituer la donnée envoyée. L'utilisation de la couche de modélisation permettrait de spécifier ce type de comportement (voir la propriété P_3bis du programme E.1 en annexe E).

Les transformations à mettre en œuvre dans ce contexte seront donc nettement plus complexes que celles vues dans ce manuscrit.

D'autres types de transformations similaires pourraient également porter sur d'autres types de variables auxiliaires, comme celles pouvant permettre de comptabiliser le nombre de cycles utilisés pour un transfert. Par exemple, nous avons remarqué que la règle de transformation T_2 suppose la connaissance du nombre de cycles X qui sépare l'envoi de l'adresse de l'envoi des données. Cette règle a par exemple été utilisée dans la section 6.2.2.3 pour un transfert simple pour une valeur de X égale à 1. Pour des transferts plus complexes, pouvant inclure des attentes, le nombre de cycles pourrait être évalué par un calcul dans la couche de modélisation.

Étant donné que le processus de raffinement proposé permet la génération d'assertions conformes à la syntaxe de PSL, il serait intéressant de fournir ces assertions (notamment celles niveau RTL) en entrée d'un outil de vérification formelle de *Model Checking* dans un processus de vérification formelle des fonctionnalités critiques.

La transformation proposée dans ce travail est un raffinement d'assertions. Il serait aussi intéressant de comparer cette démarche à celle préconisée par exemple dans [BFG⁺12] dans laquelle le modèle RTL est abstrait au niveau TLM afin de pouvoir inclure plus facilement les fonctionnalités des composants réutilisés (IPs). Dans ce contexte, il pourrait être possible de revérifier les mêmes assertions TLM sur le modèle abstrait à partir de l'implémentation RTL.

Les preuves de la transformation formelle des règles

A.1 Preuve de la règle de transformation T_3

La formule de départ est $\varphi \text{ before! } \psi$. La formule $\varphi \text{ before! } \psi \text{ and next!}[X]\psi$ est le résultat de l'intégration de la contrainte temporelle $C : \text{next!}[X]\psi$. Encore une fois ici, on utilise les définitions de l'opérateur before! et $\text{next!}[X]$ pour obtenir la formule suivante : i.e., $v \models (\neg\psi \text{ until! } \varphi \wedge \neg\psi) \text{ and } v^{X..} \models \psi$ i.e., $\exists k < |v| \text{ s.t. } v^{k..} \models (\varphi \wedge \neg\psi) \text{ and } \forall j < k, v^{j..} \models \neg\psi \text{ and } v^{X..} \models \psi$

On a d'une part, $v^{k..} \models \neg\psi$ et d'autre part $v^{X..} \models \psi$. Par conséquent, on peut déduire que $k < X$, et on obtient :

$$\exists k < X \text{ s.t. } v^{k..} \models (\varphi \wedge \neg\psi) \text{ and } \forall j < k < X, v^{j..} \models \neg\psi \text{ and } v^{X..} \models \psi$$

En remplaçant les définitions sémantiques par les opérateurs correspondants, la formule devient :

$$\text{i.e., } v \models \text{next_e!}[0..X-1](\varphi \wedge \neg\psi) \text{ and next_a!}[0..X-1](\neg\psi) \text{ and next!}[X](\psi)$$

La simplification des expressions $\text{next_e!}[0..X-1]\neg\psi$ et $\text{next_a!}[0..X-1](\neg\psi)$ nous permet d'obtenir :

$$\text{i.e., } v \models \text{next_e!}[0..X-1](\varphi) \text{ and next_a!}[0..X-1](\neg\psi) \text{ and next!}[X](\psi)$$

A.2 Preuve de la règle de transformation T_4

L'insertion de la contrainte temporelle conduit à :

$$\begin{aligned} & v \models \varphi \text{ until! } \psi \text{ and } v \models \text{next! } \psi, \text{ i.e., } v \models \varphi \text{ until! } \psi \text{ and } v^{1..} \models \psi \\ & \text{i.e., } (\exists k < |v| \text{ s.t. } v^{k..} \models \psi \text{ and } \forall j < k, v^{j..} \models \varphi) \text{ and } v^{1..} \models \psi \\ & \text{i.e., } v^{1..} \models \psi \text{ and } v^{0..} \models \varphi \end{aligned}$$

i.e., $v \models \varphi \text{ and } \text{next!}\psi$

A.3 Preuve de la règle de transformation T_5

Encore une fois, la prise en compte de la contrainte temporelle conduit à :

$v \models \varphi \text{ until! } \psi \text{ and } v \models \text{next!}[X]\psi,$
 i.e., $v \models \varphi \text{ until! } \psi \text{ and } v^{X..} \models \psi$
 i.e., $(\exists k < |v| \text{ s.t. } v^{k..} \models \psi \text{ and } \forall j < k, v^{j..} \models \varphi) \text{ and } v^{X..} \models \psi$
 i.e., $v^{0..} \models \varphi \text{ and } v^{1..} \models \varphi \text{ and } \dots v^{X-1..} \models \varphi \text{ and } v^{X..} \models \psi$
 i.e., $v \models \text{next_a!}[0..X-1](\varphi) \text{ and } \text{next!}[X](\psi)$

A.4 Preuve de la règle de transformation T_6

Ramener la formule ver le sous ensemble simple de PSL passe par sa transformation en SERE :

$v \models \{\varphi ; \text{true}[* (X-1)] ; \psi\} \mapsto \zeta$
 i.e., $\forall j / v^{0..j} \models \{\varphi ; \text{true}[* (X-1)] ; \psi\}, v^{j..} \models \zeta$
 i.e., $v^{0..X} \models \{\varphi ; \text{true}[* (X-1)] ; \psi\} \rightarrow v^{X..} \models \zeta$

 i.e., $(v^0 \models \varphi \text{ and } v^{1..X-1} \models \text{true}[* (X-1)] \text{ and } v^X \models \psi) \rightarrow v^{X..} \models \zeta$
 i.e., $(v^0 \models \varphi \text{ and } v^X \models \psi) \rightarrow v^{X..} \models \zeta$
 i.e., $v^0 \models \varphi \rightarrow (v^X \models \psi \rightarrow v^{X..} \models \zeta)$
 i.e., $v \models \varphi \rightarrow \text{next!}[X](\psi \rightarrow \zeta)$

Détails sur le Flot de raffinement

B.1 Modèle du fichier XML des règles de raffinement

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="xs3p.xsl"?>
<!--
    To change this template, choose Tools / Templates
    and open the template in the editor.
-->
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">
  <xs:element name="rules">
    <xs:annotation>
      <xs:documentation>"rules" is the root element containing
        all the transformation rules.
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="rule" type="TransformationRule"
          maxOccurs="unbounded">
          <xs:annotation>
            <xs:documentation>"rule" is the child of the root element
              .
              It contains the definition of the transformation
                rule.
            </xs:documentation>
          </xs:annotation>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
    <xs:unique name="uniqueID">
      <xs:selector xpath="xs:rule"/>
```



```

        <xs:field xpath="@id"/>
    </xs:unique>
</xs:element>

<!--complex types definition -->
<xs:complexType name="TransformationRule">
    <xs:sequence>
        <xs:element name="source" type="Rule">
            <xs:annotation>
                <xs:documentation>"source" is the left side rule
                                of the transformation rule.
            </xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element name="dest" type="Rule">
        <xs:annotation>
            <xs:documentation>"dest" is the right side rule
                            of the transformation rule.
        </xs:documentation>
    </xs:annotation>
    </xs:element>
    <xs:element name="constraint" type="xs:string"/>
</xs:sequence>
    <xs:attribute name="id" type="xs:long" use="required"/>
</xs:complexType>

<xs:complexType name="Rule">
    <xs:sequence>
        <xs:element name="text_rule" type="xs:string"/>
    </xs:sequence>
</xs:complexType>

</xs:schema>

```

PROGRAMME B.1 – Schéma XSD du fichier XML des règles de transformation

B.2 Illustration détaillée du flot de raffinement

1. Analyse des propriétés transactionnelles et extraction des fonctions de communication.
2. Choix des fonctions de communication à raffiner.
3. Génération des fichiers Excel pour l'identification des correspondances structurelles.
4. Transformation des variables globales (si elles existent).
5. Génération des chronogrammes des fonctions de communication à raffiner
6. Génération des contraintes temporelles à partir des chronogrammes.
7. Instantiation partielle des règles de transformation en prenant en compte les contraintes temporelles
8. Unification des règles de transformation partiellement instanciées avec les propriétés transactionnelles.
9. Application des substitutions obtenues sur les règles de transformation et application de ces règles de transformation sur les propriétés transactionnelles. Cette étape marque la fin de processus de raffinement temporel des assertions.
10. Dédution des correspondances structurelles à partir des données des fichiers Excel.
11. Transformation structurelle des propriétés raffinées temporellement.
12. Génération des assertions raffinées et des fichiers d'instrumentation associés.

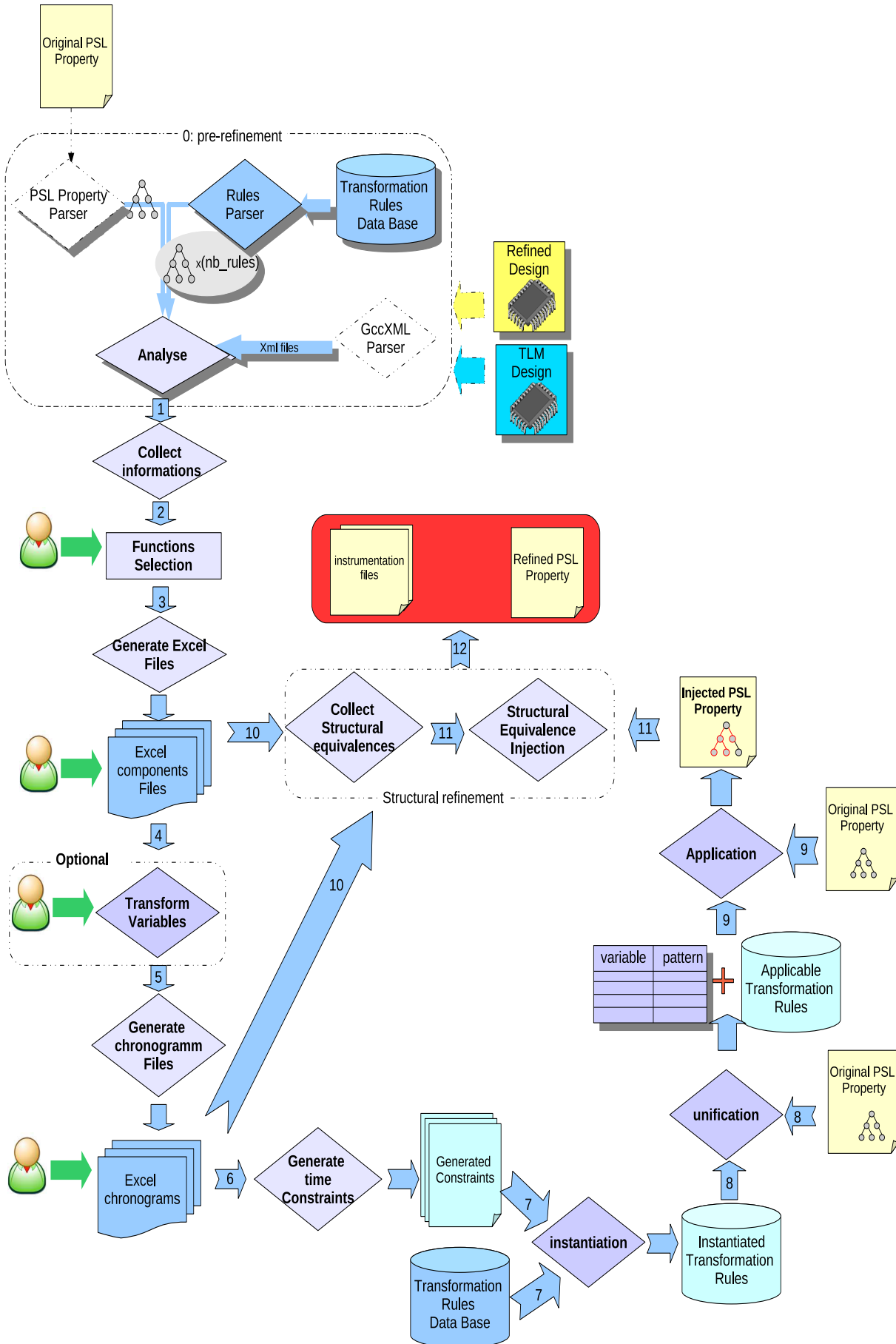


FIGURE B.1 – Flot complet détaillé

Résultats Expérimentations : Plateforme de décodage vidéo

C.1 Plateforme transactionnelle

C.1.1 Fichier d'instrumentation

Le programme C.1 contient le fichier d'instrumentation de la première assertion au niveau TLM.

Le composant `generic_noc_inst_0` est le bus. Il est déclaré dans une balise de type `needed` car le bus n'est pas observé dans l'assertion P_1 . Il est utilisé pour pouvoir accéder à la fonction `get_memory_map()`.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE instrumentation SYSTEM "instrumentation_dtd.dtd">

<instrumentation>
  <property>tlm_proprdac</property>
  <links>
    <link>
      <component>top_inst_0/eu_inst_0</component>
      <variable>eu</variable>
    </link>
    <link>
      <component>top_inst_0/ramdac_inst_0</component>
      <variable>rdac</variable>
    </link>
    <needed>
      <component>top_inst_0/generic_noc_inst_0</component>
      <variable>noc</variable>
    </needed>
  </links>
</instrumentation>
```

```
</links>
</instrumentation>
```

PROGRAMME C.1 – Fichier d'instrumentation de la propriété transactionnelle P_1 M-JPEG

C.1.2 Extraits des traces de simulation

L'extrait de la trace suivante montre le fonctionnement des moniteurs des assertions P_1 et P_2 sur la trace d'exécution de la plateforme transactionnelle. Le moniteur de la première propriété est nommé `prop_rdac_MJPEG_TLM_p0`. Celui de la seconde propriété est `prop_rdac2_MJPEG_TLM2_p0`. Les noms des moniteurs sont précédés par le nom du *Wrapper*.

Pour cette plateforme, nous avons choisi d'instrumenter la plateforme avec les deux assertions à la fois. Les deux moniteurs sont actifs sur les mêmes fonctions de communication.

Les sections commençant par `==>` correspondent aux sorties du moniteur global de l'assertion. Elles sont suivies par la liste évaluée des paramètres de l'assertion. Les sections délimitées par des pointillées (`---`) sont des affichages relatifs à la plateforme non instrumentée permettant de situer la sortie du moniteur dans le scénario d'exécution.

```
-----
--> MEMORY MAPPING RAMDAC: 3217887032 to 3217891127 - 0x1000 (4096)
-----

-----
First write operation
-----

-----
eu_inst_0 : write data 256 into address 3217887032 at 219858 ns
-----

==> 219858 ns: prop_rdac_MJPEG_TLM_p0.prop_rdac_MJPEG_TLM_p0 Valid    = TRUE <===
==> 219858 ns: prop_rdac_MJPEG_TLM_p0.prop_rdac_MJPEG_TLM_p0 Checking = FALSE <===
    eu__write_CALL = 1
    eu__write__p1 = 3217887032
    for_rdac = 1
    req_data = 256
    rdac__write_CALL = 0
    rdac__write__p2 = 0
```

```

==> 219858 ns: prop_rdac2_MJPEG_TLM_p0.prop_rdac2_MJPEG_TLM2_p0 Valid    = TRUE <==
==> 219858 ns: prop_rdac2_MJPEG_TLM_p0.prop_rdac2_MJPEG_TLM2_p0 Checking = FALSE <==
    eu__write_CALL = 1
    eu__write___p1 = 3217887032
    req_count = 1
    rdac__write_CALL = 0
    write_count = 0

```

```

-----
ramdac_inst_0 : write data 256 into address 0  at 219860 ns
-----

```

```

==> 219860 ns: prop_rdac_MJPEG_TLM_p0.prop_rdac_MJPEG_TLM_p0 Valid    = TRUE <==
==> 219860 ns: prop_rdac_MJPEG_TLM_p0.prop_rdac_MJPEG_TLM_p0 Checking = TRUE <==
    eu__write_CALL = 0
    eu__write___p1 = 3217887032
    for_rdac = 0
    req_data = 256
    rdac__write_CALL = 1
    rdac__write___p2 = 256

```

```

==> 219860 ns: prop_rdac2_MJPEG_TLM_p0.prop_rdac2_MJPEG_TLM2_p0 Valid    = TRUE <==
==> 219860 ns: prop_rdac2_MJPEG_TLM_p0.prop_rdac2_MJPEG_TLM2_p0 Checking = TRUE <==
    eu__write_CALL = 0
    eu__write___p1 = 3217887032
    req_count = 1
    rdac__write_CALL = 1
    write_count = 1

```

```

-----
Second write operation
-----

```

```

-----
eu_inst_0 : write data 144 into address 3217887032  at 220105 ns
-----

```

```

==> 220105 ns: prop_rdac_MJPEG_TLM_p0.prop_rdac_MJPEG_TLM_p0 Valid    = TRUE <==
==> 220105 ns: prop_rdac_MJPEG_TLM_p0.prop_rdac_MJPEG_TLM_p0 Checking = FALSE <==
    eu__write_CALL = 1
    eu__write___p1 = 3217887032
    for_rdac = 1
    req_data = 144

```

```

rdac__write_CALL = 0
rdac__write__p2 = 256

===> 220105 ns: prop_rdac2_MJPEG_TLM_p0.prop_rdac2_MJPEG_TLM2_p0 Valid    = TRUE <===
===> 220105 ns: prop_rdac2_MJPEG_TLM_p0.prop_rdac2_MJPEG_TLM2_p0 Checking = FALSE <===
    eu__write_CALL = 1
    eu__write__p1 = 3217887032
    req_count = 2
    rdac__write_CALL = 0
    write_count = 1

-----
ramdac_inst_0 : write data 144 into address 0  at 220107 ns
-----

===> 220107 ns: prop_rdac_MJPEG_TLM_p0.prop_rdac_MJPEG_TLM_p0 Valid    = TRUE <===
===> 220107 ns: prop_rdac_MJPEG_TLM_p0.prop_rdac_MJPEG_TLM_p0 Checking = TRUE  <===
    eu__write_CALL = 0
    eu__write__p1 = 3217887032
    for_rdac = 0
    req_data = 144
    rdac__write_CALL = 1
    rdac__write__p2 = 144

===> 220107 ns: prop_rdac2_MJPEG_TLM_p0.prop_rdac2_MJPEG_TLM2_p0 Valid    = TRUE <===
===> 220107 ns: prop_rdac2_MJPEG_TLM_p0.prop_rdac2_MJPEG_TLM2_p0 Checking = TRUE  <===
    eu__write_CALL = 0
    eu__write__p1 = 3217887032
    req_count = 2
    rdac__write_CALL = 1
    write_count = 2

```

Nous nous intéressons en premier au moniteur `prop_rdac_MJPEG_TLM_p0`.

La trace d'exécution montre deux écritures successives faites par le processeur EU0 à destination du RAMDAC. Pour la première écriture, la requête du processeur est envoyée à 219858 ns. L'adresse d'écriture correspond au début de l'espace d'adresse réservé pour le RAMDAC (3217887032). Cette requête permet d'activer le moniteur `prop_rdac_MJPEG_TLM_p0`. La trace de ce moniteur montre qu'à cet instant la variable `for_rdac` est mise à '1' pour indiquer le début d'une opération d'écriture à destination du RAMDAC. La donnée est alors mémorisée dans la variable `req_data`. À l'instant 219860 ns, le ramdac indique qu'il reçoit une requête d'écriture à l'adresse 0 (adresse obtenue après le décodage d'adresses). L'opération d'écriture active encore une fois le moniteur de l'assertion P_1 . À cet instant les sorties du moniteur global sont `Valid = TRUE` et

`Checking = TRUE` pour indiquer la satisfaction de l’assertion sur la trace d’exécution. En effet, les données écrites dans le RAMDAC correspondent à celles mémorisées dans la variable `req_data`.

Pour le moniteur de l’assertion P_2 , l’activation se fait sur les mêmes fonctions de communication. Le moniteur se charge de comparer le nombre de requêtes d’écriture adressées au RAMDAC par le processeur EU0 au nombre de requêtes d’écriture exécutées par le RAMDAC afin de s’assurer de la non perte de données entre ces deux composants. La trace d’exécution du moniteur montre que la requête envoyée à l’instant 219860 ns correspond à la première requête d’écriture envoyée par EU0. La valeur de la variable `req_count` est alors incrémentée. Elle prend la valeur 1. Cette requête est reçue par le RAMDAC à l’instant 219860 ns. À cet instant la valeur de la variable `write_count` est égale à celle de `req_count` et l’assertion est validée. Le même enchainement est repris pour la deuxième opération d’écriture qui débute à l’instant 220105 ns.

C.2 Plateforme Raffinée

C.2.1 Étapes de raffinement

Dans la suite, certaines étapes du processus de raffinement de la propriété P_1 sont décrites. Après les étapes de pré-raffinement, L’utilisateur sélectionne les fonctions concernées par le raffinement. L’assertion doit être au même niveau de granularité que la plateforme raffinée. Comme le montre la Figure C.1, les deux fonctions de communication qui passent par le bus doivent être raffinées.

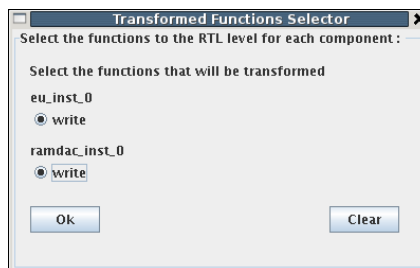


FIGURE C.1 – Interface de sélection des méthodes à transformer

Pour chaque composant, un fichier Excel est généré. Ce fichier est la base du raffinement structurel et temporel. Le premier tableau Excel contient les correspondances entre les fonctions à raffiner et les signaux de la plateforme raffinée. La Figure C.2 illustre les signaux correspondant à l’appel de la fonction `write` du composant `eu_0` et de ses paramètres. Ces signaux sont extraits de l’interface du composant au niveau signal.

La transformation des fonctions de communication selon le protocole Wishbone est uniquement structurelle. Les assertions raffinées sont le résultat du remplacement des

Component equivalence in RTL design for TLM Method:		2014/06/15 18:23:42	
	Variable Name	Component Name	Component type
TLM	eu	eu_inst_0	eu
RTL		eu_inst_0	eu
NB: if no changes don't fill cells.			
Give concerned signals for each method			
CLOCK	clk		
Edge(posedge/negedge)			
Input	Type	RTL signals	
write_CALL()	public	top_inst_0/eu_CYC ; top_inst_0/eu_WE; top_inst_0/eu_STB	
write.p1	unsigned int addr	top_inst_0/eu_ADR_O	

FIGURE C.2 – Correspondance entre les signaux et la fonction write et ses paramètres

méthodes et de leurs paramètres par les signaux correspondants dans la plateforme raffinée. Les fichiers d'instrumentation des propriétés raffinées sont très ressemblants en raison des similitudes entre les deux assertions de départ.

C.2.2 Fichier d'instrumentation généré

Le programme C.2 contient le fichier d'instrumentation généré par l'outil de raffinement pour la propriété P_1 raffinée. Le fichier relatif à l'assertion P_2 est identique.

Le fichier montre qu'au niveau RTL, l'observation du moniteur de la propriété `rtl_proprdac` se fait sur l'horloge. Ce composant est alors définie dans la balise `link`. Les signaux `eu_STB` et `rdac_STB` sont déclarés dans des balises du type `needed_rose_fell` car la fonction `rose` est appelée sur ces signaux afin de permettre la détection des fronts montants sur ces signaux. Les autres éléments (signaux et composants) déclarés comme étant "needed" sont uniquement accessibles en lecture.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE instrumentation PUBLIC "SYSTEM" "instrumentation_dtd.dtd">
<instrumentation><property><![CDATA[rtl_proprdac]]></property>
  <links>
    <link>
      <component>clk</component>
      <variable>clk</variable>
    </link>
    <needed>
      <component>top_inst_0/generic_noc_inst_0</component>
      <variable>noc</variable>
    </needed>
    <needed>
```

```

    <component>top_inst_0/eu0_CYC</component>
    <variable>eu_CYC</variable>
</needed>
<needed>
    <component>top_inst_0/eu0_WE</component>
    <variable>eu_WE</variable>
</needed>
<needed>
    <component>top_inst_0/eu0_DATA</component>
    <variable>eu_DAT</variable>
</needed>
<needed>
    <component>top_inst_0/eu0_ADDR</component>
    <variable>eu_ADR</variable>
</needed>
<needed>
    <component>top_inst_0/CYC</component>
    <variable>rdac_CYC</variable>
</needed>
<needed>
    <component>top_inst_0/WE</component>
    <variable>rdac_WE</variable>
</needed>
<needed>
    <component>top_inst_0/data_w</component>
    <variable>rdac_DAT</variable>
</needed>

<needed_rose_fell>
    <component>top_inst_0/eu0_STB</component>
    <variable>eu_STB</variable>
</needed_rose_fell>
<needed_rose_fell>
    <component>top_inst_0/STB</component>
    <variable>rdac_STB</variable>
</needed_rose_fell>
</links>
</instrumentation>

```

PROGRAMME C.2 – Fichier d'instrumentation de la propriété P_1 du MJPEG

C.2.3 Extrait des traces de simulation

Dans la plateforme raffinée, nous avons utilisé les fonctions de la bibliothèque `sc_trace` de SystemC afin de générer les chronogrammes des signaux de la plateforme au cours de l'exécution de la plateforme instrumentée. Nous avons extrait, dans la Figure C.3, la

partie du chronogramme des signaux de communication entre le EU_0 et le RAMDAC. Nous soulignons que pour cette traces reportées par la suite, la configuration mémoire du RAMDAC est la suivante : --> MEMORY MAPPING RAMDAC0: 135756756 to 135760851.

L'extrait montre les deux premières opérations d'écriture initiées par le processeur à destination du RAMDAC. Le chronogramme montre que le démarrage d'une opération d'écriture est caractérisé par le passage des signaux STB à '1'. Ceci est valable aussi bien pour le maître (le processeur EU0) que pour l'esclave (le RAMDAC). Le transfert entre le processeur et le RAMDAC nécessite un cycle d'horloge. Ainsi, le cycle suivant l'envoi de la première requête d'écriture (eu0_data= 256), le signal rdac_STB du RAMDAC passe à '1' lui indiquant qu'il est la cible d'une requête du bus. La nature de l'opération est désignée par le signal rdac_WE.

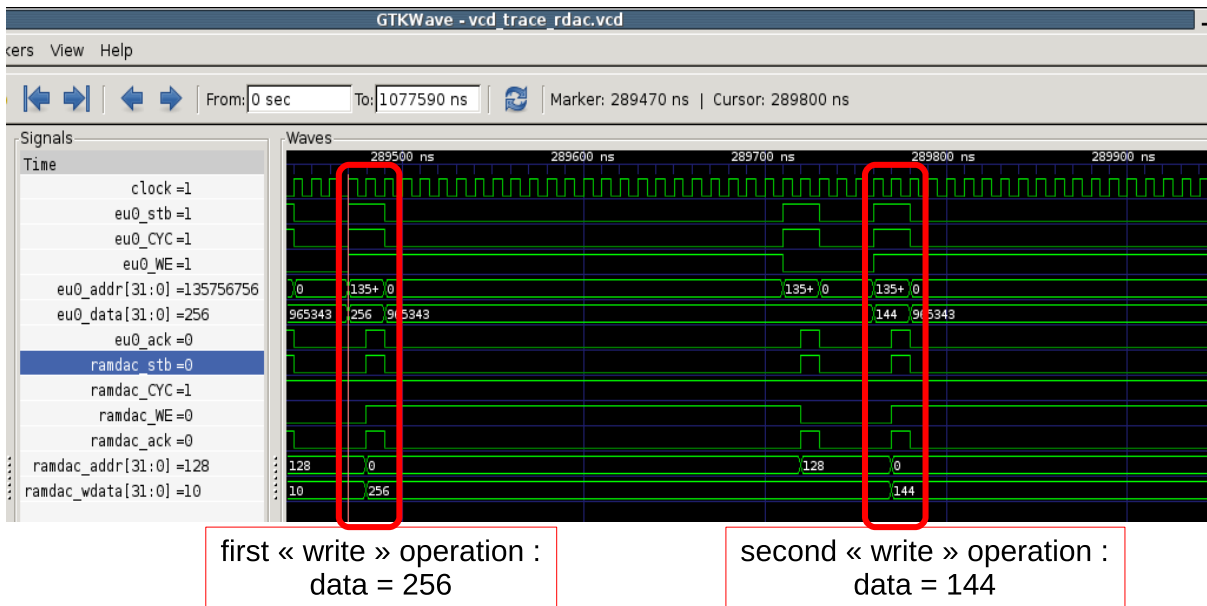


FIGURE C.3 – Extrait la trace d'exécution en utilisant Gtkwave

Comme pour la plateforme transactionnelle, nous avons choisi d'instrumenter la plateforme RTL par les deux moniteurs générés en utilisant l'outil ISIS. L'extrait de la trace montré ci-dessous contient les deux premières opérations d'écritures destinées au RAMDAC. Les extraits choisis correspondent aux mêmes opérations reportées au niveau transactionnel. L'activation des moniteurs de surveillance se fait sur les fronts montants de l'horloge (clk__rising_CALL = 1).

L'espace d'adressage réservée au RAMDAC est donnée au début de la trace. Le moniteur de l'assertion P_1 raffinée est proprdac_MJPEG_RTL_p0. proprdac2_MJPEG_RTL_p0 est celui de la propriété P_2 raffinée.

```
-----
--> MEMORY MAPPING RAMDAC0: 135756756 to 135760851 - 0x1000 (4096)
-----
```

 First write operation

```

===> 289480 ns: proprdac_MJPEG_RTL_p0.proprdac_MJPEG_RTL_p0 Valid    = TRUE <===
===> 289480 ns: proprdac_MJPEG_RTL_p0.proprdac_MJPEG_RTL_p0 Checking = FALSE <===
      clk__rising_CALL = 1
      eu_STB->rose()    = 1
      eu_CYC = 1
      eu_WE = 1
      req_addr = 135756756
      for_rdac = 1
      req_data = 256
      rdac_STB->rose() = 0
      rdac_WE = 0
      rdac_CYC = 1
      rdac_data = 10

===> 289480 ns: proprdac_MJPEG_RTL_p0.proprdac2_MJPEG_RTL_p0 Valid    = TRUE <===
===> 289480 ns: proprdac_MJPEG_RTL_p0.proprdac2_MJPEG_RTL_p0 Checking = FALSE <===
      clk__rising_CALL = 255
      eu_STB->rose()    = 1
      eu_CYC = 1
      eu_WE = 1
      req_addr = 135756756
      req_count = 1
      rdac_STB->rose() = 0
      rdac_CYC = 1
      rdac_WE = 0
      write_count = 0

===> 289490 ns: proprdac_MJPEG_RTL_p0.proprdac_MJPEG_RTL_p0 Valid    = TRUE <===
===> 289490 ns: proprdac_MJPEG_RTL_p0.proprdac_MJPEG_RTL_p0 Checking = TRUE  <===
      clk__rising_CALL = 1
      eu_STB->rose()    = 0
      eu_CYC = 1
      eu_WE = 1
      req_addr = 135756756
      for_rdac = 1
      req_data = 256
      rdac_STB->rose() = 1
      rdac_WE = 1
      rdac_CYC = 1
      rdac_data = 256

```

```

====> 289490 ns: proprdac_MJPEG_RTL_p0.proprdac2_MJPEG_RTL_p0 Valid    = TRUE  <===
====> 289490 ns: proprdac_MJPEG_RTL_p0.proprdac2_MJPEG_RTL_p0 Checking = TRUE  <===
      clk__rising_CALL = 255
      eu_STB->rose()    = 1
      eu_CYC = 1
      eu_WE = 1
      req_addr = 135756756
      req_count = 1
      rdac_STB->rose() = 1
      rdac_CYC = 1
      rdac_WE = 1
      write_count = 1

```

.....

```

-----
second write operation
-----

```

```

====> 289770 ns: proprdac_MJPEG_RTL_p0.proprdac_MJPEG_RTL_p0 Valid    = TRUE  <===
====> 289770 ns: proprdac_MJPEG_RTL_p0.proprdac_MJPEG_RTL_p0 Checking = FALSE <===
      clk__rising_CALL = 1
      eu_STB->rose()    = 1
      eu_CYC = 1
      eu_WE = 1
      req_addr = 135756756
      for_rdac = 1
      req_data = 144
      rdac_STB->rose() = 0
      rdac_WE = 0
      rdac_CYC = 1
      rdac_data = 256

```

```

====> 289770 ns: proprdac_MJPEG_RTL_p0.proprdac2_MJPEG_RTL_p0 Valid    = TRUE  <===
====> 289770 ns: proprdac_MJPEG_RTL_p0.proprdac2_MJPEG_RTL_p0 Checking = FALSE <===
      clk__rising_CALL = 255
      eu_STB->rose()    = 1
      eu_CYC = 1
      eu_WE = 1
      req_addr = 135756756
      req_count = 2
      rdac_STB->rose() = 0
      rdac_CYC = 1
      rdac_WE = 0
      write_count = 1

```

```

==> 289780 ns: proprdac_MJPEG_RTL_p0.proprdac_MJPEG_RTL_p0 Valid    = TRUE <===
==> 289780 ns: proprdac_MJPEG_RTL_p0.proprdac_MJPEG_RTL_p0 Checking = TRUE <===
      clk__rising_CALL = 1
      eu_STB->rose()   = 0
      eu_CYC = 1
      eu_WE = 1
      req_addr = 135756756
      for_rdac = 1
      req_data = 144
      rdac_STB->rose() = 1
      rdac_WE = 1
      rdac_CYC = 1
      rdac_data = 144

==> 289780 ns: proprdac_MJPEG_RTL_p0.proprdac2_MJPEG_RTL_p0 Valid    = TRUE <===
==> 289780 ns: proprdac_MJPEG_RTL_p0.proprdac2_MJPEG_RTL_p0 Checking = TRUE <===
      clk__rising_CALL = 255
      eu_STB->rose()   = 1
      eu_CYC = 1
      eu_WE = 1
      req_addr = 135756756
      req_count = 2
      rdac_STB->rose() = 1
      rdac_CYC = 1
      rdac_WE = 1
      write_count = 2

```

La première écriture commence à 289480 ns. À cet instant, les moniteurs repèrent le front montant survenu sur le signal `eu_STB`. La variable `for_rdac` du premier moniteur (`proprdac_MJPEG_RTL_p0`) passe à '1' pour indiquer le début d'une requête d'écriture destinée au ramdac. Simultanément, la variable `req_count` du second moniteur est incrémentée.

La requête atteint le RAMDAC au cycle d'horloge suivant (à 289490 ns). Les moniteurs détectent le passage à '1' du signal `rdac_STB` du RAMDAC. Pour le premier moniteur, la donnée recue par le RAMDAC (`rdac_data`) est égale à celle envoyée par le processeur et mémorisée dans la variable `req_data`. Pour le second moniteur, l'arrivée de la requête d'écriture au RAMDAC permet d'incrémenter la valeur de `write_count` qui devient égale à celle de la variable `req_count`. Les deux moniteurs indiquent alors que les assertions respectives sont satisfaites.

Résultats Expérimentations : Plateforme de traitement d'images

D.1 Plateforme transactionnelle

D.1.1 Propriété P_3

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE instrumentation SYSTEM "instrumentation_dtd.dtd">
<instrumentation>
  <property>prop1</property>
  <links>
    <link>
      <component>platform/dma_a</component>
      <variable>dma_a</variable>
    </link>
    <link>
      <component>platform/m_leon_a</component>
      <variable>leon_a</variable>
    </link>
  </links>
</instrumentation>
```

PROGRAMME D.1 – Fichier d'instrumentation généré pour la propriété P_3 raffinée

Ce qui suit est un extrait de la race d'exécution de la plateforme instrumentée en utilisant le moniteur de cette propriété généré en utilisant l'outil ISIS. Le moniteur est activée à chaque appel de la fonction `write()` du `leon_a` et à chaque génération d'une interruption par le `dma_a` (appel de `generate_irq()`).

La vérification se la formule démarre à l'instant 36 ns, quand le `leon_a` configure le registre `ctrl_reg` du `dma_a`. Ainsi, l'adresse de l'écriture est égale à celle du `ctrl_reg` et

la valeur écrite est égale à celle de `start`. Le `dma_a` démarre alors le transfert demandé (état 'TRANSFERRING').

```

====> 36 ns: prop1_P1_TML_CODES_p0.prop1_P1_TML_CODES_p0 Valid    = TRUE <===
====> 36 ns: prop1_P1_TML_CODES_p0.prop1_P1_TML_CODES_p0 Checking = FALSE<===
====> 36 ns: prop1_P1_TML_CODES_p0.prop1_P1_TML_CODES_p0 Pending  = FALSE<===
        leon_a___write_block_CALL = 1
        leon_a___write_block___p1 = 16777228
        ctrl_reg = 16777228
        value = 1
        start_transfer = 1
        dma_a___generate_irq_CALL = 0

```

```

-----
Bus_a @ 36 ns DEBUG: Incoming address was 16777228
DMA_a @ 36 ns DEBUG: Written 1 to DMA_CONTROL
DMA_a @ 36 ns DEBUG: Internal state is now 'TRANSFERRING'
-----

```

....

```

-----
DMA_a @ 384048 ns DEBUG: Internal state is now 'IDLE'
-----

```

```

====> 384048 ns: prop1_P1_TML_CODES_p0.prop1_P1_TML_CODES_p0 Valid    = TRUE <===
====> 384048 ns: prop1_P1_TML_CODES_p0.prop1_P1_TML_CODES_p0 Checking = FALSE<===
====> 384048 ns: prop1_P1_TML_CODES_p0.prop1_P1_TML_CODES_p0 Pending  = TRUE <===
        leon_a___write_block_CALL = 0
        leon_a___write_block___p1 = 16777228
        ctrl_reg = 16777228
        value = 1
        start_transfer = 1
        dma_a___generate_irq_CALL = 1

```

....

```

====> 700068 ns: prop1_P1_TML_CODES_p0.prop1_P1_TML_CODES_p0 Valid    = TRUE <===
====> 700068 ns: prop1_P1_TML_CODES_p0.prop1_P1_TML_CODES_p0 Checking = TRUE <===
====> 700068 ns: prop1_P1_TML_CODES_p0.prop1_P1_TML_CODES_p0 Pending  = FALSE<===
        leon_a___write_block_CALL = 1
        leon_a___write_block___p1 = 16777228
        ctrl_reg = 16777228
        value = 1
        start_transfer = 1
        dma_a___generate_irq_CALL = 0

```

```

-----
Bus_a @ 700068 ns DEBUG: Incoming address was 16777228

```

```
DMA_a @ 700068 ns DEBUG: Written 0x00000001 to DMA_CONTROL
DMA_a @ 700068 ns DEBUG: Internal state is now 'TRANSFERRING'
-----
```

Le moniteur consiste à vérifier que l'interruption de fin de transfert survient avant la prochaine configuration du `dma_a`.

Dans l'extrait de la trace, l'interruption est envoyée à 384048 ns alors que la deuxième configuration a lieu à 700068 ns. L'assertion est alors validée sur cette trace. Les sorties du moniteur global affichent `Valid = TRUE` et `Checking = TRUE`.

D.1.2 Propriété P_4

Dans l'extrait de la trace, la première configuration de la FFT faite par le `leon_b` a lieu à 850296 ns. Cette opération démarre la vérification de la formule de l'assertion. La taille de l'image est mémorisée dans la variable `fft_size`. La FFT passe à l'état 'READING' pour récupérer l'image de la mémoire `meme_b`.

```
-----
Bus_b @ 850272 ns DEBUG: Incoming address was 33554432
FFT @ 850284 ns DEBUG: Written 8000 to WRITE_ADDR
FFT @ 850284 ns DEBUG: Writing is now active
-----

====> 850296 ns: prop4_P4_TML_CODES_p0.prop4_P4_TML_CODES_p0 Valid    = TRUE <====
====> 850296 ns: prop4_P4_TML_CODES_p0.prop4_P4_TML_CODES_p0 Checking = FALSE<====
====> 850296 ns: prop4_P4_TML_CODES_p0.prop4_P4_TML_CODES_p0 Pending  = FALSE<====
      leon_b__write_block__p1 = 33554432
      leon_b__read_block__p1 = 0
      leon_b__write_block_CALL = 1
      a_read_addr = 33554432
      leon_b__read_block_END = 0
      a_write_length = 33554444
      write_length_value = 0
      fft_size = 2000

-----

FFT @ 850308 ns DEBUG: Reading is now active
FFT @ 850308 ns DEBUG: FFT block is in 'READING' state
-----

.....
```

```
-----
FFT @ 1050300 ns DEBUG: Read 2000 from WRITE_LENGTH
-----
```

```
====> 1050312 ns: prop4_P4_TML_CODES_p0.prop4_P4_TML_CODES_p0 Valid    = TRUE <===
====> 1050312 ns: prop4_P4_TML_CODES_p0.prop4_P4_TML_CODES_p0 Checking = FALSE<===
====> 1050312 ns: prop4_P4_TML_CODES_p0.prop4_P4_TML_CODES_p0 Pending  = FALSE<===
      leon_b___write_block___p1 = 33554432
      leon_b___read_block___p1  = 33554444
      leon_b___write_block_CALL = 0
      a_read_addr = 33554432
      leon_b___read_block_END  = 1
      a_write_length = 33554444
      write_length_value = 2000
      fft_size = 2000

....
====> 1605776 ns: prop4_P4_TML_CODES_p0.prop4_P4_TML_CODES_p0 Valid    = TRUE <===
====> 1605776 ns: prop4_P4_TML_CODES_p0.prop4_P4_TML_CODES_p0 Checking = TRUE <===
====> 1605776 ns: prop4_P4_TML_CODES_p0.prop4_P4_TML_CODES_p0 Pending  = FALSE<===
      leon_b___write_block___p1 = 33554432
      leon_b___read_block___p1  = 16777232
      leon_b___write_block_CALL = 1
      a_read_addr = 33554432
      leon_b___read_block_END  = 0
      a_write_length = 33554444
      write_length_value = 2000
      fft_size = 2000
```

```
-----
FFT @ 1606364 ns DEBUG: Reading is now active
FFT @ 1606364 ns DEBUG: FFT block is in 'READING' state
-----
```

L'assertion consiste à vérifier que la lecture du registre `a_write_length` par le `leon_b` ait lieu avant la prochaine configuration de la FFT. La trace montre que la fin de l'opération de lecture initiée par le `leon_b` vers le registre `a_write_length` a lieu à 1050312 ns. La donnée lue correspond à la valeur de `fft_size` (2000). La deuxième configuration de la FFT a lieu à 1605776 ns. À cet instant l'assertion est satisfaite sur cette trace.

D.1.3 Propriété P_5

Cet extrait de trace permet de vérifier l'assertion sur le premier paquet de donnée présent dans l'IO. La lecture de ce dernier (*packet 0*) doit s'achever avant l'arrivée du prochain paquet (*packet 1*).

Le premier paquet est présent dans l'IO à l'instant 0s (`io_module___generate_irq_CALL = '1'`).

Au niveau TLM, la lecture d'un paquet entier de donnée est atomique. Cette opération a lieu à 144048 ns (`dma_a___read_block_END = '1'`). L'adresse donnée est celle du début du bloc de données. La moniteur indique que la trace est satisfaite à l'instant 700 us , quand le deuxième paquet est disponible au niveau de l'IO.

```
-----
IO @ 0 s INFO:  Packet n°0 is now ready to be read
-----
```

```
====> 0 s: prop5_P5_TML_CODES_p0.prop5_P5_TML_CODES_p0 Valid    = TRUE <===
====> 0 s: prop5_P5_TML_CODES_p0.prop5_P5_TML_CODES_p0 Checking = FALSE<===
====> 0 s: prop5_P5_TML_CODES_p0.prop5_P5_TML_CODES_p0 Pending  = FALSE<===
        io_module___generate_irq_CALL = 1
        dma_a___read_block_END = 0
        dma_a___read_block___p1 = 0
        io_module_address = 536870912
.....
```

```
-----
IO @ 143664 ns DEBUG: Packet n°0 was read
-----
```

```
====> 144048 ns: prop5_P5_TML_CODES_p0.prop5_P5_TML_CODES_p0 Valid    = TRUE <===
====> 144048 ns: prop5_P5_TML_CODES_p0.prop5_P5_TML_CODES_p0 Checking = FALSE<===
====> 144048 ns: prop5_P5_TML_CODES_p0.prop5_P5_TML_CODES_p0 Pending  = TRUE <===
        io_module___generate_irq_CALL = 0
        dma_a___read_block_END = 1
        dma_a___read_block___p1 = 536870912
        io_module_address = 536870912
.....
```

```
-----
IO @ 700 us INFO:  Packet n°1 is now ready to be read
-----
```

```
====> 700 us: prop5_P5_TML_CODES_p0.prop5_P5_TML_CODES_p0 Valid    = TRUE <===
====> 700 us: prop5_P5_TML_CODES_p0.prop5_P5_TML_CODES_p0 Checking = TRUE <===
====> 700 us: prop5_P5_TML_CODES_p0.prop5_P5_TML_CODES_p0 Pending  = FALSE<===
        io_module___generate_irq_CALL = 1
        dma_a___read_block_END = 0
        dma_a___read_block___p1 = 536870912
        io_module_address = 536870912
```

D.2 Raffinement selon le protocole Wishbone

D.2.1 Propriété P_3 raffinée

```

vunit rtl_prop03{

    // HDL_DECLs :
    unsigned int ctrl_reg = 16777216 + dma_reg::a_dma_control;
    unsigned int start_transfer = dma_reg::dma_control_start_mask;

    // HDL_STMTs :

    // PROPERTY :
assert
    (always (leon_a_cyc && leon_a_we && rose(leon_a_stb) &&
        grant_leon_a && leon_a_adr == ctrl_reg &&
        leon_a_data == start_transfer
            -> next (dma_a.generate_irq_CALL()
                before! (leon_a_cyc && leon_a_we &&
                    rose(leon_a_stb) &&
                    leon_a_adr == ctrl_reg &&
                    leon_a_data == start_transfer) ))
    )@ (main_clk.rising() || dma_a.generate_irq_CALL());
}

```

PROGRAMME D.2 – Propriété P_3 raffinée selon le protocole Wishbone

Trace d'exécution de la plateforme instrumentée avec le moniteur de cette assertion :

```

-----
dma_orig_inst @ 108 ns DEBUG: Written 1 to DMA_CONTROL
dma_orig_inst @ 108 ns DEBUG: Internal state is now 'TRANSFERRING'
dma_orig_inst @ 108 ns DEBUG: DMA transfer started.
    Source address: 0x20000000 - destination address: 0000000000 - length: 4000
dma_orig_inst @ 108 ns INFO: Read Block of size 16000 at address 20000000
-----

===> 120 ns: prop01_ASTRUM_WB_P1_p0.prop01_ASTRUM_WB_P1_p0 Valid    = TRUE<===
===> 120 ns: prop01_ASTRUM_WB_P1_p0.prop01_ASTRUM_WB_P1_p0 Checking = FALSE<===
===> 120 ns: prop01_ASTRUM_WB_P1_p0.prop01_ASTRUM_WB_P1_p0 Pending  = FALSE<===
    ctrl_reg = 16777228
    start_transfer = 1
    dma_a__generate_irq_CALL = 0
    main_clk__rising_CALL = 1
    leon_a_cyc = 1
    leon_a_we = 1
    leon_a_stb->rose() = 1

```

```

leon_a_adr = 16777228
leon_a_data = 1
grant_leon_a = 1
....

-----
dma_orig_inst @ 156108 ns DEBUG: END WRITE BLOCK data 0 to 0000000000
dma_orig_inst @ 156108 ns DEBUG: Internal state is now 'IDLE'
-----

==> 156108 ns: prop01_ASTRUM_WB_P1_p0.prop01_ASTRUM_WB_P1_p0 Valid    = TRUE<==
==> 156108 ns: prop01_ASTRUM_WB_P1_p0.prop01_ASTRUM_WB_P1_p0 Checking = FALSE<==
==> 156108 ns: prop01_ASTRUM_WB_P1_p0.prop01_ASTRUM_WB_P1_p0 Pending  = FALSE<==
ctrl_reg = 16777228
start_transfer = 1
dma_a__generate_irq_CALL = 1
main_clk__rising_CALL = 0
leon_a_cyc = 0
leon_a_we = 0
leon_a_stb->rose() = 0
leon_a_adr = 0
leon_a_data = 1
grant_leon_a = 0
.....

-----
dma_orig_inst @ 700116 ns DEBUG: Written 0x00000001 to DMA_CONTROL
dma_orig_inst @ 700116 ns DEBUG: Internal state is now 'TRANSFERRING'
dma_orig_inst @ 700116 ns DEBUG: DMA transfer started. Source address:
    0x20000000 - destination address: 0000000000 - length: 4000
dma_orig_inst @ 700116 ns INFO: Read Block of size 16000 at address 20000000
-----

==> 700128 ns: prop01_ASTRUM_WB_P1_p0.prop01_ASTRUM_WB_P1_p0 Valid    = TRUE<==
==> 700128 ns: prop01_ASTRUM_WB_P1_p0.prop01_ASTRUM_WB_P1_p0 Checking = TRUE<==
==> 700128 ns: prop01_ASTRUM_WB_P1_p0.prop01_ASTRUM_WB_P1_p0 Pending  = FALSE<==
ctrl_reg = 16777228
start_transfer = 1
dma_a__generate_irq_CALL = 0
main_clk__rising_CALL = 1
leon_a_cyc = 1
leon_a_we = 1
leon_a_stb->rose() = 1
leon_a_adr = 16777228
leon_a_data = 1
grant_leon_a = 1

```

L'extrait de la trace montre que la première configuration du `dma_a` par la `leon_a` a lieu à 120 ns. Le début de l'écriture dans le registre de contrôle se caractérise par le passage à '1' du signal `leon_a_stb`. À la fin du transfert, le `dma_a` envoie une requête (`dma_a___generate_irq_CALL='1'`) à 156108 ns. Le `dma_a` passe à l'état 'IDLE'. La deuxième configuration du `dma_a` se fait à ns près la fin du premier transfert. L'assertion est alors validée sur cette trace.

D.2.2 Propriété P_4 raffinée

```
vunit rtl_prop04{

    // HDL_DECLS :
    unsigned long a_read_addr = 33554432 + fft_reg::a_read_addr;
    unsigned long a_read_length = 33554432 + fft_reg::a_read_length;
    unsigned long a_write_length = 33554432 + fft_reg::
        a_write_length;
    unsigned int fft_size;

    // HDL_STMTs :
    if(rose(leon_b_stb) && leon_b_cyc &&
        leon_b_we && grant_leon_b &&
        leon_b_adr == a_read_length && main_clk.rising())
        fft_size = leon_b_data_w;

    // PROPERTY :
assert
    (always
        (rose(leon_b_stb) && leon_b_cyc &&
            leon_b_we && grant_leon_b &&
            leon_b_adr == a_read_addr
        =>next( (leon_b_cyc && !leon_b_we && leon_b_stb &&
            grant_leon_b &&
            leon_b_adr == a_write_length &&
            leon_b_data_r == fft_size && bus_b_ack == 1)
                before! ((rose(leon_b_stb) && leon_b_cyc &&
                    leon_b_we && grant_leon_b &&
                    leon_b_adr == a_read_addr) )
            )
        )
    )@ (main_clk.rising());
}
```

PROGRAMME D.3 – Propriété P_4 raffinée selon le protocole Wishbone

Trace d'exécution de la plateforme instrumentée avec le moniteur de cette assertion :

Leon_b_ configure the FFT "a_read_length" register
the transfer length is putten into the fft_size variable

fft_orig_inst @ 454848 ns DEBUG: Written 2000 to READ_LENGTH

```
====> 454860 ns: prop04_ASTRUM_WB_P4_p0.prop04_ASTRUM_WB_P4_p0 Valid    = TRUE  <===
====> 454860 ns: prop04_ASTRUM_WB_P4_p0.prop04_ASTRUM_WB_P4_p0 Checking = FALSE <===
====> 454860 ns: prop04_ASTRUM_WB_P4_p0.prop04_ASTRUM_WB_P4_p0 Pending  = FALSE <===
```

```
    a_read_addr = 33554432
    a_write_length = 33554444
    fft_size = 2000
    main_clk__rising_CALL = 1
    leon_b_data_w = 2000
    leon_b_cyc = 1
    leon_b_we = 1
    leon_b_stb->rose() = 1
    leon_b_adr = 33554436
    leon_b_data_r = 0
    grant_leon_b  = 1
```

....

Leon_b_ configure the FFT "a_read_addr" register :

fft_orig_inst @ 454920 ns DEBUG: Written 0000000000 to READ_ADDR

fft_orig_inst @ 454920 ns DEBUG: Reading is now active

fft_orig_inst @ 454920 ns DEBUG: FFT block is in 'READING' state

fft_orig_inst @ 454920 ns DEBUG: Reading 16 words at address 0000000000

```
====> 454932 ns: prop04_ASTRUM_WB_P4_p0.prop04_ASTRUM_WB_P4_p0 Valid    = TRUE  <===
====> 454932 ns: prop04_ASTRUM_WB_P4_p0.prop04_ASTRUM_WB_P4_p0 Checking = FALSE <===
====> 454932 ns: prop04_ASTRUM_WB_P4_p0.prop04_ASTRUM_WB_P4_p0 Pending  = FALSE <===
```

```
    a_read_addr = 33554432
    a_write_length = 33554444
    fft_size = 2000
    main_clk__rising_CALL = 1
    leon_b_data_w = 0
    leon_b_cyc = 1
    leon_b_we = 1
    leon_b_stb->rose() = 1
    leon_b_adr = 33554432
    leon_b_data_r = 0
```



```
grant_leon_b = 1
```

```
====> 454944 ns: prop04_ASTRIMUM_WB_P4_p0.prop04_ASTRIMUM_WB_P4_p0 Valid    = TRUE  <====
====> 454944 ns: prop04_ASTRIMUM_WB_P4_p0.prop04_ASTRIMUM_WB_P4_p0 Checking = FALSE <====
====> 454944 ns: prop04_ASTRIMUM_WB_P4_p0.prop04_ASTRIMUM_WB_P4_p0 Pending  = TRUE  <====
      a_read_addr = 33554432
      a_write_length = 33554444
      fft_size = 2000
      main_clk__rising_CALL = 1
      leon_b_data_w = 0
      leon_b_cyc = 1
      leon_b_we = 1
      leon_b_stb->rose() = 0
      leon_b_adr = 33554432
      leon_b_data_r = 0
      grant_leon_b = 1
```

```
.....
```

```
-----
Leon_b_ reads the  FFT "a_write_length" register
the read value is into "leon_b_data_r" variable
leon_b_data_r ==  fft_size
-----
```

```
====> 585420 ns: prop04_ASTRIMUM_WB_P4_p0.prop04_ASTRIMUM_WB_P4_p0 Valid    = TRUE  <====
====> 585420 ns: prop04_ASTRIMUM_WB_P4_p0.prop04_ASTRIMUM_WB_P4_p0 Checking = FALSE <====
====> 585420 ns: prop04_ASTRIMUM_WB_P4_p0.prop04_ASTRIMUM_WB_P4_p0 Pending  = FALSE <====
      a_read_addr = 33554432
      a_write_length = 33554444
      fft_size = 2000
      main_clk__rising_CALL = 1
      leon_b_data_w = 0
      leon_b_cyc = 1
      leon_b_we = 0
      leon_b_stb->rose() = 1
      leon_b_adr = 33554444
      leon_b_data_r = 2000
      grant_leon_b = 1
```

```
.....
```

```
-----
Leon_b_ configure the next FFT transfert
( writes into "a_read_addr" register)
```

```

fft_orig_inst @ 1154892 ns DEBUG: Written 0000000000 to READ_ADDR
fft_orig_inst @ 1154892 ns DEBUG: Reading is now active
fft_orig_inst @ 1154892 ns DEBUG: FFT block is in 'READING' state
fft_orig_inst @ 1154892 ns DEBUG: Reading 16 words at address 0000000000
-----

==> 1154904 ns: prop04_ASTRIUM_WB_P4_p0.prop04_ASTRIUM_WB_P4_p0 Valid    = TRUE  <==
==> 1154904 ns: prop04_ASTRIUM_WB_P4_p0.prop04_ASTRIUM_WB_P4_p0 Checking = TRUE  <==
==> 1154904 ns: prop04_ASTRIUM_WB_P4_p0.prop04_ASTRIUM_WB_P4_p0 Pending = FALSE <==
      a_read_addr = 33554432
      a_write_length = 33554444
      fft_size = 2000
      main_clk__rising_CALL = 1
      leon_b_data_w = 0
      leon_b_cyc = 1
      leon_b_we = 1
      leon_b_stb->rose = 1
      leon_b_adr = 33554432
      leon_b_data_r = 0
      grant_leon_b  = 1

```

Le moniteur de cette assertion est actif sur tous les fronts montants de l'horloge `main_clk`. Le front d'horloge ayant lieu à 454860 ns concide avec la première configuration du registre `a_read_length` de la FFT par le processeur `leon_b`. La variable `fft_size` prend alors le contenu du signal `leon_b_data_w`. Ensuite, à 454932 ns, `leon_b` configure le registre `a_read_addr` et la lecture de la FFT commence. Le moniteur passe à l'état *Pending* en attente de la satisfaction de l'opérande droit de l'opérateur **before!** (dans sa version forte). À l'instant 585420 ns, le `leon_b` fini la lecture du registre `a_write_length`. La valeur lue est égale au contenu de `fft_size`. La sortie *Pending* du moniteur global passe alors à 'FALSE'. La prochaine lecture de la FFT démarre à 1154904 ns, après le début de traitement du `leon_b`. L'assertion est alors satisfaite à cet instant.

D.2.3 Propriété P_5 raffinée

La modélisation d'un transfert rafale Wishbone se fait en utilisant le signal `CTI_0/I`¹. Dans la plateforme raffinée, nous avons implémenté une rafale à adresses incrémentales (*Incrementing burst cycle*, `CTI_0` = '010'). Le démarrage d'un transfert rafale est semblable à celui d'un transfert simple. Il est caractérisé par le passage du signal `STB_0/I` à '1'. La

¹Des informations additionnelles telles que le type du transfert rafale peuvent être données en utilisant le signal `BTE_0/I`.

fin de ce transfert se fait par l'envoi d'une séquence particulière dite *End-Of-Burst* sur le signal CTI_0/I (CTI_0 = '111'). En se référant à ces données, la transformation de l'assertion P_5 se fait ainsi :

```
vunit rtl_prop05{

    // HDL_DECLs :
    const unsigned long io_module_address = 536870912;
    //added after generation
    const unsigned int packet_size = 16000;
    bool read_start;

    // HDL_STMTs :
    if( dma_a_HADR == io_module_address &&
        !dma_a_HWRITE && dma_a_HTRANS==2 &&
        dma_a_HBURST==7 && dma_a_HSIZE==2 ) {
        read_started = true;
    }
    if( dma_a_HADR == io_module_address+packet_size &&
        !dma_a_HWRITE && dma_a_HTRANS==3 &&
        dma_a_HBURST==7 && dma_a_HSIZE==2 ) {
        read_started = false;
    }

    // PROPERTY :
assert
    (always
        ( io_module.generate_irq_CALL()
          =>next((read_start &&
                dma_a_cyc && !dma_a_we && dma_a_stb &&
                dma_a_cti == '111' &&
                (dma_a_adr == (io_module_address+(packet_size)-4)))
              before! (io_module.generate_irq_CALL() ) )
        )
    )@ (main_clk.rising() || io_module.generate_irq_CALL());
}
```

PROGRAMME D.4 – Propriété P_5 raffinée selon le protocole Wishbone

Trace d'exécution de la plateforme instrumentée avec le moniteur de cette assertion :

```
-----
io_handler_orig_inst @ 0 s INFO:  Packet N°0 is now ready to be read
-----
```

```
====> 0 s: prop05_ASTRUM_WB_P5_p0.prop05_ASTRUM_WB_P5_p0  Valid    = TRUE <===
====> 0 s: prop05_ASTRUM_WB_P5_p0.prop05_ASTRUM_WB_P5_p0  Checking = FALSE<===
====> 0 s: prop05_ASTRUM_WB_P5_p0.prop05_ASTRUM_WB_P5_p0  Pending  = FALSE<===
        io_module__generate_irq_CALL = 1
```

```

        io_module_address = 536870912
        main_clk__rising_CALL = 0
        read_started = 0
        dma_a_cyc = 0
        dma_a_we = 0
        dma_a_stb = 0
        dma_a_adr = 0
        grant_dma_a = 0
        dma_a_cti = 000

```

.....

```

-----
Start transfert to IO module (dma_a_adr == io_module_address)
leon_a_orig_inst @ 132 ns INFO: Started DMA_a transfer
-----

```

```

====> 144 ns: prop05_ASTRUM_WB_P5_p0.prop05_ASTRUM_WB_P5_p0 Valid    = TRUE  <===
====> 144 ns: prop05_ASTRUM_WB_P5_p0.prop05_ASTRUM_WB_P5_p0 Checking = FALSE <===
====> 144 ns: prop05_ASTRUM_WB_P5_p0.prop05_ASTRUM_WB_P5_p0 Pending  = TRUE  <===
        io_module__generate_irq_CALL = 0
        io_module_address = 536870912
        main_clk__rising_CALL = 1
        read_started = 1
        dma_a_cyc = 1
        dma_a_we = 0
        dma_a_stb = 1
        dma_a_adr = 536870912
        grant_dma_a = 1
        dma_a_cti = 010

```

....

```

-----
End of the transfert to IO module
(dma_a_adr == io_module_address+16000-4)
-----

```

```

====> 54120 ns: prop05_ASTRUM_WB_P5_p0.prop05_ASTRUM_WB_P5_p0 Valid    = TRUE  <===
====> 54120 ns: prop05_ASTRUM_WB_P5_p0.prop05_ASTRUM_WB_P5_p0 Checking = FALSE <===
====> 54120 ns: prop05_ASTRUM_WB_P5_p0.prop05_ASTRUM_WB_P5_p0 Pending  = FALSE <===
        io_module__generate_irq_CALL = 0
        io_module_address = 536870912
        main_clk__rising_CALL = 1
        read_started = 1
        dma_a_cyc = 1

```

```

dma_a_we = 0
dma_a_stb = 1
dma_a_adr = 536886908
grant_dma_a = 1
dma_a_cti = 111

```

```

-----
dma_orig_inst @ 54120 ns DEBUG: END READ BLOCK data 0
-----

```

```

....

```

```

-----
io_handler_orig_inst @ 700 us INFO: Packet N°1 is now ready to be read
-----

```

```

==> 700 us: prop05_ASTRUM_WB_P5_p0.prop05_ASTRUM_WB_P5_p0 Valid    = TRUE <==
==> 700 us: prop05_ASTRUM_WB_P5_p0.prop05_ASTRUM_WB_P5_p0 Checking = TRUE <==
==> 700 us: prop05_ASTRUM_WB_P5_p0.prop05_ASTRUM_WB_P5_p0 Pending  = FALSE<==
    io_module__generate_irq_CALL = 1
    io_module_address = 536870912
    main_clk__rising_CALL = 0
    read_started = 0
    dma_a_cyc = 0
    dma_a_we = 0
    dma_a_stb = 0
    dma_a_adr = 0
    grant_dma_a = 0
    dma_a_cti = 000

```

L'IO indique la présence du premier paquet à 0 ns (`io_module__generate_irq_CALL = '1'`). Suite à la configuration du `dma_a`, celui-ci commence le transfert du paquet de donnée au front d'horloge survenu à 144 ns. L'adresse de la lecture (`dma_a_adr`) est alors égale au début de l'espace d'adressage de l'IO. Le moniteur reste à l'état `textitPending` jusqu'à la fin de la lecture du paquet entier. La lecture du paquet se termine à 54120 ns, avant la réception du paquet de données suivant. L'assertion est validée à 700 us suite à la notification de l'IO (`io_module__generate_irq_CALL = '1'`).

D.3 Raffinement selon le protocole AHB

D.3.1 Propriété P_3 raffinée

La Figure D.1 illustre la première feuille du fichier Excel correspondant au composant Leon_a. La figure montre l'ensemble des signaux de la plateforme raffinée impliqués dans l'opération d'écriture simple représentée par la méthode "write_block" et ces paramètres au niveau transactionnel.

Component equivalence in RTL design for TLM		2014/07/01 17:01:02	
	Variable Name	Component Name	Component type
TLM	leon_a	platform/m_leon_a	leon_a *
RTL		platform/m_leon_a	leon_a *
NB: if no changes don't fill cells.			
Give concerned signals for each method			
CLOCK	clk		
Edge(posedge/negedge)			
Input	Type	RTL signals	
write_block_CALL()	public	platform/leon_a_HWRITE;platform/leon_a_HGRANT;platform/leon_a_HTRANS;platform/leon_a_HBURST;platform/leon_a_HSIZE;platform/bus_a_HREADY	
write_block.p2	unsigned char *	platform/leon_a_HWDATA	
write_block.p1	const long unsigned int	platform/leon_a_HADDR	

FIGURE D.1 – Correspondance entre la fonction write_block et ses paramètres et les signaux AHB

Ce extrait illustre la sortie de l'outil de raffinement lors du raffinement de l'assertion P_3 . La trace du l'outil a été volontairement enrichie afin de montrer explicitement les résultats intermédiaires. La trace commence à partir de l'étape de l'unification simple (ou filtrage). Elle montre l'identification de l'ensemble des règles applicables sur l'assertion d'origine et donne les substitutions obtenues pour chaque règle de cet ensemble.

```
* * UNIFY RULES WITH PROPERTY...
* * Done

* * Rules To Apply:
    1) Rule 4 : (VAR_a) before! (VAR_b && VAR_c)
    ---> ((VAR_b =>next! [VAR_T] (!VAR_c)) until! VAR_a)
    && next_event! (VAR_a) (VAR_b =>next! [VAR_T] (!VAR_c))
    2) Rule 6 : (VAR_a && VAR_b) => (VAR_c)
    ---> VAR_a =>next! [VAR_T] (VAR_b => VAR_c)
* * Done.

* * APPLY SUBSTITUTIONS ON RULES...
```

Apply Transformation on Rule :

```
L: (VAR_a) before! (leon_a.write_block_CALL() &&
  leon_a.write_block.p1== VAR_p1 &&
  leon_a.write_block.p2== VAR_p0)
  --->

R: (((leon_a.write_block_CALL() && leon_a.write_block.p1 == VAR_p1
  =>next!(!(leon_a.write_block.p2 == VAR_p0))))
  until!(VAR_a))
  && next_event!(VAR_a)((leon_a.write_block_CALL() &&
leon_a.write_block.p1 == VAR_p1
  =>next!(!(leon_a.write_block.p2 == VAR_p0))))
```

```
Substitution is :{
  {VAR_a|->dma_a.generate_irq_CALL()}
  {VAR_p1|->ctrl_reg}
  {VAR_p0|->start_transfer}
}
```

Apply Transformation on Rule :

```
L: (leon_a.write_block_CALL() &&
  leon_a.write_block.p1== VAR_p1 &&
  leon_a.write_block.p2== VAR_p0) => (VAR_c)
  --->

R: (leon_a.write_block_CALL() && leon_a.write_block.p1 == VAR_p1
  =>next!((leon_a.write_block.p2 == VAR_p0
  =>VAR_c)))
```

```
Substitution :{
{VAR_p1|->ctrl_reg}
{VAR_p0|->start_transfer}
  {VAR_c|->next((((leon_a.write_block_CALL() && leon_a.write_block.p1 == ctrl_reg
  =>next!(!(leon_a.write_block.p2 == start_transfer))))
  until!(dma_a.generate_irq_CALL()))
  && next_event!(dma_a.generate_irq_CALL())((leon_a.write_block_CALL() &&
    leon_a.write_block.p1 == ctrl_reg
    =>next!(!(leon_a.write_block.p2 == start_transfer))))))
}
```

* * Done.

* * APPLY TRANSFORMATION RULES ON PROPERTY "prop1"...

* * Done.

* * INJECT INPUTS SIGNALS INTO PROPERTY "prop1"...

* * Done.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE instrumentation PUBLIC "SYSTEM" "instrumentation_dtd.dtd">
<instrumentation>
  <property><![CDATA[rtl_prop1]]></property>
  <links>
    <link>
      <component>platform/dma_orig_inst</component>
      <variable>dma_a</variable>
    </link>
    <link>
      <component>clk</component>
      <variable>HCLK</variable>
    </link>
    <needed>
      <component>platform/leon_a_HWRITE</component>
      <variable>leon_a_HWRITE</variable>
    </needed>
    <needed>
      <component>platform/leon_a_HTRANS</component>
      <variable>leon_a_HTRANS</variable>
    </needed>
    <needed>
      <component>platform/leon_a_HBURST</component>
      <variable>leon_a_HBURST</variable>
    </needed>
    <needed>
      <component>platform/leon_a_HSIZE</component>
      <variable>leon_a_HSIZE</variable>
    </needed>
    <needed>
      <component>platform/leon_a_HWDATA</component>
      <variable>leon_a_HWDATA</variable>
    </needed>
    <needed>
      <component>platform/leon_a_HADDR</component>
      <variable>leon_a_HADDR</variable>
    </needed>
    <needed_rose_fell>
      <component>platform/leon_a_HGRANT</component>
      <variable>leon_a_HGRANT</variable>
    </needed_rose_fell>
    <needed_rose_fell>
      <component>platform/bus_a_HREADY</component>
      <variable>bus_a_HREADY</variable>
    </needed_rose_fell>
  </links>
</instrumentation>
```


</links>

</instrumentation>

PROGRAMME D.5 – Fichier d'instrumentation généré pour la propriété P_3 raffinée

L'extrait de trace donné ci-dessous commence à l'instant 132 ns quand le `Leon_a` procède à la première configuration du registre `ctrl_reg` du `dma_a`. L'assertion passe à l'état *Pending*. Le transfert commence au cycle suivant, quand le `leon_a` écrit la valeur '1' (START) dans le registre de contrôle. À partir du cycle suivant (à 156 ns), les sorties *Checking* et *Pending* du moniteur sont maintenues actives. La sortie *Pending* repasse à FALSE dès que toutes les obligations futur des opérateurs **until!** et **next_event** sont satisfaites. C'est le cas à l'instant 300120 ns suite à la notification de l'appel de la fonction `generate_irq`. Le passage de la sortie *Checking* à FALSE tout en ayant *Valid* à TRUE indique que le moniteur n'est plus actif et qu'aucune violation n'a été détectée. En conclusion, la trace satisfait l'assertion.

Leon_a configures the DMA ctrl_reg to start a new transfer

```

====> 132 ns: rtl_prop1_PROP1_RTL_THESE_p0.rtl_prop1_PROP1_RTL_THESE_p0 Valid      = TRUE <====
====> 132 ns: rtl_prop1_PROP1_RTL_THESE_p0.rtl_prop1_PROP1_RTL_THESE_p0 Checking = FALSE <====
====> 132 ns: rtl_prop1_PROP1_RTL_THESE_p0.rtl_prop1_PROP1_RTL_THESE_p0 Pending  = TRUE  <====
      ctrl_reg = 16777228
      start_transfer = 1
      dma_a___generate_irq_CALL = 0
      HCLK__rising_CALL = 1
      leon_a_HWRITE = 1
      leon_a_HGRANT = 1
      leon_a_HTRANS = 2
      leon_a_HBURST = 0
      leon_a_HSIZE = 2
      leon_a_HREADY = 1
      leon_a_HADR = 16777228
      leon_a_HWDATA = 4000
      leon_a_HREADY->prev(1) = 1
      leon_a_HGRANT->prev(1) = 1

```

Leon_a writes START into the control register

```

leon_a_orig_inst @ 132 ns INFO: Started DMA_a transfer
dma_orig_inst @ 132 ns DEBUG: Internal state is now 'TRANSFERRING'
dma_orig_inst @ 132 ns DEBUG: DMA transfer started. Source address:
      0x20000000 - destination address: 0000000000 - length: 4000

```

```

====> 144 ns: rtl_prop1_PROP1_RTL_THESE_p0.rtl_prop1_PROP1_RTL_THESE_p0 Valid    = TRUE  <===
====> 144 ns: rtl_prop1_PROP1_RTL_THESE_p0.rtl_prop1_PROP1_RTL_THESE_p0 Checking = FALSE <===
====> 144 ns: rtl_prop1_PROP1_RTL_THESE_p0.rtl_prop1_PROP1_RTL_THESE_p0 Pending  = TRUE  <===
      ctrl_reg = 16777228
      start_transfer = 1
      dma_a___generate_irq_CALL = 0
      HCLK___rising_CALL = 1
      leon_a_HWRITE = 1
      leon_a_HGRANT = 1
      leon_a_HTRANS = 3
      leon_a_HBURST = 0
      leon_a_HSIZE = 2
      leon_a_HREADY = 1
      leon_a_HADR = 0
      leon_a_HWDATA = 1
      leon_a_HREADY->prev(1) = 1
      leon_a_HGRANT->prev(1) = 1

====> 156 ns: rtl_prop1_PROP1_RTL_THESE_p0.rtl_prop1_PROP1_RTL_THESE_p0 Valid    = TRUE  <===
====> 156 ns: rtl_prop1_PROP1_RTL_THESE_p0.rtl_prop1_PROP1_RTL_THESE_p0 Checking = TRUE  <===
====> 156 ns: rtl_prop1_PROP1_RTL_THESE_p0.rtl_prop1_PROP1_RTL_THESE_p0 Pending  = TRUE  <===
      ctrl_reg = 16777228
      start_transfer = 1
      dma_a___generate_irq_CALL = 0
      HCLK___rising_CALL = 1
      leon_a_HWRITE = 0
      leon_a_HGRANT = 0
      leon_a_HTRANS = 0
      leon_a_HBURST = 0
      leon_a_HSIZE = 0
      leon_a_HREADY = 1
      leon_a_HADR = 0
      leon_a_HWDATA = 0
      leon_a_HREADY->prev(1) = 1
      leon_a_HGRANT->prev(1) = 1

====> 168 ns: rtl_prop1_PROP1_RTL_THESE_p0.rtl_prop1_PROP1_RTL_THESE_p0 Valid    = TRUE  <===
====> 168 ns: rtl_prop1_PROP1_RTL_THESE_p0.rtl_prop1_PROP1_RTL_THESE_p0 Checking = TRUE  <===
====> 168 ns: rtl_prop1_PROP1_RTL_THESE_p0.rtl_prop1_PROP1_RTL_THESE_p0 Pending  = TRUE  <===
      ctrl_reg = 16777228
      start_transfer = 1
      dma_a___generate_irq_CALL = 0
      HCLK___rising_CALL = 1
      leon_a_HWRITE = 0
      leon_a_HGRANT = 0
      leon_a_HTRANS = 0

```

```

leon_a_HBURST = 0
leon_a_HSIZE = 0
leon_a_HREADY = 1
leon_a_HADR = 0
leon_a_HWDATA = 0
leon_a_HREADY->prev(1) = 1
leon_a_HGRANT->prev(1) = 0

```

.....

END of the DMA transfer

dma_orig_inst @ 300120 ns DEBUG: Internal state is now 'IDLE'

```

===> 300120 ns: rtl_prop1_PROP1_RTL_THESE_p0.rtl_prop1_PROP1_RTL_THESE_p0 Valid    = TRUE <===
===> 300120 ns: rtl_prop1_PROP1_RTL_THESE_p0.rtl_prop1_PROP1_RTL_THESE_p0 Checking = TRUE  <===
===> 300120 ns: rtl_prop1_PROP1_RTL_THESE_p0.rtl_prop1_PROP1_RTL_THESE_p0 Pending  = FALSE <===

```

```

ctrl_reg = 16777228
start_transfer = 1
dma_a___generate_irq_CALL = 1
HCLK___rising_CALL = 0
leon_a_HWRITE = 0
leon_a_HGRANT = 0
leon_a_HTRANS = 0
leon_a_HBURST = 0
leon_a_HSIZE = 0
leon_a_HREADY = 1
leon_a_HADR = 0
leon_a_HWDATA = 0
leon_a_HREADY->prev(1) = 1
leon_a_HGRANT->prev(1) = 0

```

```

===> 300132 ns: rtl_prop1_PROP1_RTL_THESE_p0.rtl_prop1_PROP1_RTL_THESE_p0 Valid    = TRUE <===
===> 300132 ns: rtl_prop1_PROP1_RTL_THESE_p0.rtl_prop1_PROP1_RTL_THESE_p0 Checking = FALSE <===
===> 300132 ns: rtl_prop1_PROP1_RTL_THESE_p0.rtl_prop1_PROP1_RTL_THESE_p0 Pending  = FALSE <===

```

```

ctrl_reg = 16777228
start_transfer = 1
HCLK___rising_CALL = 1
dma_a___generate_irq_CALL = 0
leon_a_HWRITE = 0
leon_a_HGRANT = 0
leon_a_HTRANS = 0
leon_a_HBURST = 0
leon_a_HSIZE = 0

```

```

leon_a_HREADY = 1
leon_a_HADR = 0
leon_a_HWDATA = 0
leon_a_HREADY->prev(1) = 1
leon_a_HGRANT->prev(1) = 0

```

D.3.2 Propriété P_4 raffinée

Le raffinement concerne les deux fonctions “write_block()” et “read_block()” du leon_b. Le raffinement de l’expression leon_b.write_block_CALL() est identique à celui de leon_a.write_block_CALL() (*c.f.* figure 6.7).

Signals Evolution for method :		read_block	Clock:
signal	cycle	0	1
platform/leon_b_HWRITE			0
platform/leon_b_HTRANS			3
platform/leon_b_HBURST			0
platform/leon_b_HSIZE			2
platform/bus_b_HREADY			1
platform/bus_b_HRESP			0
platform/leon_b_HRDATA			x
platform/m_leon_b/HADDR		x	
fft_size			
read_block end condition		leon_b_HTRANS == 3	

FIGURE D.2 – Chronogramme correspondant à read_block_END (transfert simple)

Le raffinement de l’expression leon_b.read_block_END() nécessite l’application de la règle de raffinement T_2 afin d’introduire la contrainte temporelle du protocole AHB concernant les transferts simples. La Figure D.2 décrit le chronogramme des signaux impliqués dans l’opération de lecture simple. Le tableau montre que la fin de la lecture coïncide avec la phase de données du protocole AHB, quand les données sont disponibles sur le bus des données. L’envoi de l’adresse se fait au cycle précédent, à la phase d’adresse. La contrainte déduite est alors la suivante :

```

leon_b.read_block_END() && write_length_value == fft_size occurs 1 cycle after
leon_b.read_block.p1 == a_write_length.

```

Deux cas de figure se présentent pour le raffinement de cette assertion : la transformation temporelle en FL ou en SERE. Les extraits de la trace d’exécution de la plateforme raffinée instrumentée par le moniteur de chaque assertion résultante sont donnés par la suite. Les mêmes instants sont présentés afin de permettre la comparaison du comportement des deux moniteurs par rapport à la même trace. Nous soulignons que la validation

du moniteur de l'assertion SERE survient plus tôt que celle du moniteur FL. Ca dernier indique que la trace satisfait l'assertion FL à 1447440 ns au début du second transfert de la FFT. Le moniteur SERE valide la trace à 876468 ns un cycle après la fin de la lecture du registre `a_write_length` par le `leon_b`. Ce décalage d'un cycle de justifie par la présence du moniteur de composition séquentielle à la fin de la SERE ; juste avant le dernier moniteur [*] (qui accepte la trace vide).

D.3.2.1 Raffinement en FL

```

-----
Leon_b_ configure the FFT "a_read_length" register
AHB address phase
-----

===> 747408 ns: rtl_prop4_PROP4_RTL_THESE_p0.rtl_prop4_PROP4_RTL_THESE_p0 Valid    = TRUE  <===
===> 747408 ns: rtl_prop4_PROP4_RTL_THESE_p0.rtl_prop4_PROP4_RTL_THESE_p0 Checking = FALSE <===
===> 747408 ns: rtl_prop4_PROP4_RTL_THESE_p0.rtl_prop4_PROP4_RTL_THESE_p0 Pending  = FALSE <===
    a_read_addr = 33554432
    a_write_length = 33554444
    fft_size = 0
    HCLK__rising_CALL = 1
    leon_b_HWDATA = 4075945984
    leon_b_HWRITE = 1
    leon_b_HTRANS = 2
    leon_b_HBURST = 0
    leon_b_HSIZE = 2
    leon_b_HGRANT = 1
    leon_b_HREADY = 1
    leon_b_HADR = 33554436
    leon_b_HRESP = 0
    leon_b_HRDATA = 0
    leon_b_HBUSREQ->prev(1) = 1
    leon_b_HLOCK->prev(1) = 1
    leon_b_HWRITE->prev(1) = 0
    leon_b_HTRANS->prev(1) = 0
    leon_b_HBURST->prev(1) = 0
    leon_b_HSIZE->prev(1) = 0
    leon_b_HGRANT->prev(1) = 1
    leon_b_HREADY->prev(1) = 1
    leon_b_HADR->prev(1) = 0

-----

put the transfer length (leon_a_hwddata) into fft_size
AHB data phase
-----

```

```

====> 747420 ns: rtl_prop4_PROP4_RTL_THESE_p0.rtl_prop4_PROP4_RTL_THESE_p0 Valid      = TRUE  <====
====> 747420 ns: rtl_prop4_PROP4_RTL_THESE_p0.rtl_prop4_PROP4_RTL_THESE_p0 Checking = FALSE <====
====> 747420 ns: rtl_prop4_PROP4_RTL_THESE_p0.rtl_prop4_PROP4_RTL_THESE_p0 Pending  = FALSE <====
      a_read_addr = 33554432
      a_write_length = 33554444
      fft_size = 2000
      HCLK__rising_CALL = 1
      leon_b_HWDATA = 2000
      leon_b_HWRITE = 1
      leon_b_HTRANS = 3
      leon_b_HBURST = 0
      leon_b_HSIZE = 2
      leon_b_HGRANT = 1
      leon_b_HREADY = 1
      leon_b_HADR = 0
      leon_b_HRESP = 0
      leon_b_HRDATA = 0
      leon_b_HBUSREQ->prev(1) = 1
      leon_b_HLOCK->prev(1) = 1
      leon_b_HWRITE->prev(1) = 1
      leon_b_HTRANS->prev(1) = 2
      leon_b_HBURST->prev(1) = 0
      leon_b_HSIZE->prev(1) = 2
      leon_b_HGRANT->prev(1) = 1
      leon_b_HREADY->prev(1) = 1
      leon_b_HADR->prev(1) = 33554436

```

....

```

-----
Leon_b_ configure the FFT "a_read_addr" register :
      start of first transfert (AHB address phase)
-----

```

```

====> 747480 ns: rtl_prop4_PROP4_RTL_THESE_p0.rtl_prop4_PROP4_RTL_THESE_p0 Valid      = TRUE  <====
====> 747480 ns: rtl_prop4_PROP4_RTL_THESE_p0.rtl_prop4_PROP4_RTL_THESE_p0 Checking = FALSE <====
====> 747480 ns: rtl_prop4_PROP4_RTL_THESE_p0.rtl_prop4_PROP4_RTL_THESE_p0 Pending  = FALSE <====
      a_read_addr = 33554432
      a_write_length = 33554444
      fft_size = 2000
      HCLK__rising_CALL = 1
      leon_b_HWDATA = 8000
      leon_b_HWRITE = 1
      leon_b_HTRANS = 2
      leon_b_HBURST = 0
      leon_b_HSIZE = 2

```

```

leon_b_HGRANT = 1
leon_b_HREADY = 1
leon_b_HADR = 33554432
leon_b_HRESP = 0
leon_b_HRDATA = 0
leon_b_HBUSREQ->prev(1) = 1
leon_b_HLOCK->prev(1) = 1
leon_b_HWRITE->prev(1) = 0
leon_b_HTRANS->prev(1) = 0
leon_b_HBURST->prev(1) = 0
leon_b_HSIZE->prev(1) = 0
leon_b_HGRANT->prev(1) = 1
leon_b_HREADY->prev(1) = 1
leon_b_HADR->prev(1) = 0

```

```

-----
leon_b_orig_inst @ 747480 ns INFO: Started a new FFT
fft_orig_inst @ 747480 ns DEBUG: FFT block is in 'READING' state
-----

```

.....

```

-----
Leon_b_ reads the FFT "a_write_length" register
AHB address phase
-----

```

```

===> 876444 ns: rtl_prop4_PROP4_RTL_THESE_p0.rtl_prop4_PROP4_RTL_THESE_p0 Valid    = TRUE <===
===> 876444 ns: rtl_prop4_PROP4_RTL_THESE_p0.rtl_prop4_PROP4_RTL_THESE_p0 Checking = FALSE <===
===> 876444 ns: rtl_prop4_PROP4_RTL_THESE_p0.rtl_prop4_PROP4_RTL_THESE_p0 Pending  = TRUE  <===
    a_read_addr = 33554432
    a_write_length = 33554444
    fft_size = 2000
    HCLK__rising_CALL = 1
    leon_b_HWDATA = 0
    leon_b_HWRITE = 0
    leon_b_HTRANS = 2
    leon_b_HBURST = 0
    leon_b_HSIZE = 2
    leon_b_HGRANT = 1
    leon_b_HREADY = 1
    leon_b_HADR = 33554444
    leon_b_HRESP = 0
    leon_b_HRDATA = 1952

```

```

leon_b_HBUSREQ->prev(1) = 1
leon_b_HLOCK->prev(1) = 1
leon_b_HWRITE->prev(1) = 0
leon_b_HTRANS->prev(1) = 0
leon_b_HBURST->prev(1) = 0
leon_b_HSIZE->prev(1) = 0
leon_b_HGRANT->prev(1) = 1
leon_b_HREADY->prev(1) = 1
leon_b_HADR->prev(1) = 0

```

AHB data phase :

```

the read value is into "leon_b_data_r" variable
leon_b_data_r == fft_size
-----

```

```

==> 876456 ns: rtl_prop4_PROP4_RTL_THESE_p0.rtl_prop4_PROP4_RTL_THESE_p0 Valid    = TRUE  <===
==> 876456 ns: rtl_prop4_PROP4_RTL_THESE_p0.rtl_prop4_PROP4_RTL_THESE_p0 Checking = FALSE <===
==> 876456 ns: rtl_prop4_PROP4_RTL_THESE_p0.rtl_prop4_PROP4_RTL_THESE_p0 Pending  = FALSE <===
a_read_addr = 33554432
a_write_length = 33554444
fft_size = 2000
HCLK__rising_CALL = 1
leon_b_HWDATA = 0
leon_b_HWRITE = 0
leon_b_HTRANS = 3
leon_b_HBURST = 0
leon_b_HSIZE = 2
leon_b_HGRANT = 1
leon_b_HREADY = 1
leon_b_HADR = 0
leon_b_HRESP = 0
leon_b_HRDATA = 2000
leon_b_HBUSREQ->prev(1) = 1
leon_b_HLOCK->prev(1) = 1
leon_b_HWRITE->prev(1) = 0
leon_b_HTRANS->prev(1) = 2
leon_b_HBURST->prev(1) = 0
leon_b_HSIZE->prev(1) = 2
leon_b_HGRANT->prev(1) = 1
leon_b_HREADY->prev(1) = 1
leon_b_HADR->prev(1) = 33554444

```

....


```

-----
Leon_b_ configure the next FFT transfert (second transfert)
( writes the transfer length into "a_read_addr" register)
-----

```

```

==> 1447440 ns: rtl_prop4_PROP4_RTL_THESE_p0.rtl_prop4_PROP4_RTL_THESE_p0 Valid    = TRUE <==
==> 1447440 ns: rtl_prop4_PROP4_RTL_THESE_p0.rtl_prop4_PROP4_RTL_THESE_p0 Checking = TRUE <==
==> 1447440 ns: rtl_prop4_PROP4_RTL_THESE_p0.rtl_prop4_PROP4_RTL_THESE_p0 Pending  = FALSE <==
    a_read_addr = 33554432
    a_write_length = 33554444
    fft_size = 2000
    HCLK__rising_CALL = 1
    leon_b_HWDATA = 8000
    leon_b_HWRITE = 1
    leon_b_HTRANS = 2
    leon_b_HBURST = 0
    leon_b_HSIZE = 2
    leon_b_HGRANT = 1
    leon_b_HREADY = 1
    leon_b_HADR = 33554432
    leon_b_HRESP = 0
    leon_b_HRDATA = 0
    leon_b_HBUSREQ->prev(1) = 1
    leon_b_HLOCK->prev(1) = 1
    leon_b_HWRITE->prev(1) = 0
    leon_b_HTRANS->prev(1) = 0
    leon_b_HBURST->prev(1) = 0
    leon_b_HSIZE->prev(1) = 0
    leon_b_HGRANT->prev(1) = 1
    leon_b_HREADY->prev(1) = 1
    leon_b_HADR->prev(1) = 0

```

```

-----
leon_b_orig_inst @ 1447440 ns INFO: Started a new FFT
fft_orig_inst @ 1447440 ns DEBUG: FFT block is in 'READING' state
-----

```

D.3.2.2 Raffinement en SERE

```

-----
Leon_b_ configure the FFT "a_read_length" register
AHB address phase
-----

```

```

====> 747408 ns: prop4_P4_RTL_CODES_p0.prop4_P4_RTL_CODES_p0 Valid      = TRUE  <====
====> 747408 ns: prop4_P4_RTL_CODES_p0.prop4_P4_RTL_CODES_p0 Checking = FALSE <====
====> 747408 ns: prop4_P4_RTL_CODES_p0.prop4_P4_RTL_CODES_p0 Pending  = FALSE <====

```

```

    a_read_addr = 33554432
    a_write_length = 33554444
    fft_size = 0
    HCLK___rising_CALL = 1
    leon_b_wdata = 4075945984
    grant_leon_b = 1
    leon_b_hwrite = 1
    leon_b_adr = 33554436
    leon_b_htrans = 2
    leon_b_hburst = 0
    bus_b_hready = 1
    leon_b_rdata = 0
    grant_leon_b->prev(1) = 1
    leon_b_hwrite->prev(1) = 0
    leon_b_adr->prev(1) = 0
    leon_b_htrans->prev(1) = 0
    leon_b_hburst->prev(1) = 0
    bus_b_hready->prev(1) = 1

```

```

-----
put the transfer length (leon_a_hwdata) into fft_size
AHB data phase
-----

```

```

====> 747420 ns: prop4_P4_RTL_CODES_p0.prop4_P4_RTL_CODES_p0 Valid      = TRUE  <====
====> 747420 ns: prop4_P4_RTL_CODES_p0.prop4_P4_RTL_CODES_p0 Checking = FALSE <====
====> 747420 ns: prop4_P4_RTL_CODES_p0.prop4_P4_RTL_CODES_p0 Pending  = FALSE <====

```

```

    a_read_addr = 33554432
    a_write_length = 33554444
    fft_size = 2000
    HCLK___rising_CALL = 1
    leon_b_wdata = 2000
    grant_leon_b = 1
    leon_b_hwrite = 1
    leon_b_adr = 0
    leon_b_htrans = 3
    leon_b_hburst = 0
    bus_b_hready = 1
    leon_b_rdata = 0
    grant_leon_b->prev(1) = 1
    leon_b_hwrite->prev(1) = 1
    leon_b_adr->prev(1) = 33554436

```

```
leon_b_htrans->prev(1) = 2
leon_b_hburst->prev(1) = 0
bus_b_hready->prev(1) = 1
```

```
-----
Leon_b_ configure the FFT "a_read_addr" register :
  start of first transfert (AHB address phase)
-----
```

```
====> 747480 ns: prop4_P4_RTL_CODES_p0.prop4_P4_RTL_CODES_p0 Valid    = TRUE  <===
====> 747480 ns: prop4_P4_RTL_CODES_p0.prop4_P4_RTL_CODES_p0 Checking = FALSE <===
====> 747480 ns: prop4_P4_RTL_CODES_p0.prop4_P4_RTL_CODES_p0 Pending  = FALSE <===
      a_read_addr = 33554432
      a_write_length = 33554444
      fft_size = 2000
      HCLK__rising_CALL = 1
      leon_b_wdata = 8000
      grant_leon_b = 1
      leon_b_hwrite = 1
      leon_b_adr = 33554432
      leon_b_htrans = 2
      leon_b_hburst = 0
      bus_b_hready = 1
      leon_b_rdata = 0
      grant_leon_b->prev(1) = 1
      leon_b_hwrite->prev(1) = 0
      leon_b_adr->prev(1) = 0
      leon_b_htrans->prev(1) = 0
      leon_b_hburst->prev(1) = 0
      bus_b_hready->prev(1) = 1
```

```
-----
Leon_b reads the a_write_length register
AHB address phase
-----
```

```
====> 876444 ns: prop4_P4_RTL_CODES_p0.prop4_P4_RTL_CODES_p0 Valid    = TRUE  <===
====> 876444 ns: prop4_P4_RTL_CODES_p0.prop4_P4_RTL_CODES_p0 Checking = FALSE <===
====> 876444 ns: prop4_P4_RTL_CODES_p0.prop4_P4_RTL_CODES_p0 Pending  = TRUE  <===
      a_read_addr = 33554432
      a_write_length = 33554444
      fft_size = 2000
      HCLK__rising_CALL = 1
      leon_b_wdata = 0
```

```

grant_leon_b = 1
leon_b_hwrite = 0
leon_b_adr = 33554444
leon_b_htrans = 2
leon_b_hburst = 0
bus_b_hready = 1
leon_b_rdata = 1952
grant_leon_b->prev(1) = 1
leon_b_hwrite->prev(1) = 0
leon_b_adr->prev(1) = 0
leon_b_htrans->prev(1) = 0
leon_b_hburst->prev(1) = 0
bus_b_hready->prev(1) = 1

```

```

-----
The read value (leon_b_hrdata) is equal to fft_size
AHB data phase
-----

```

```

Evaluation 1 pending at 876456 ns
==> 876456 ns: prop4_P4_RTL_CODES_p0.prop4_P4_RTL_CODES_p0 Valid = TRUE <===
==> 876456 ns: prop4_P4_RTL_CODES_p0.prop4_P4_RTL_CODES_p0 Checking = FALSE <===
==> 876456 ns: prop4_P4_RTL_CODES_p0.prop4_P4_RTL_CODES_p0 Pending = TRUE <===
a_read_addr = 33554432
a_write_length = 33554444
fft_size = 2000
HCLK__rising_CALL = 1
leon_b_wdata = 0
grant_leon_b = 1
leon_b_hwrite = 0
leon_b_adr = 0
leon_b_htrans = 3
leon_b_hburst = 0
bus_b_hready = 1
leon_b_rdata = 2000
grant_leon_b->prev(1) = 1
leon_b_hwrite->prev(1) = 0
leon_b_adr->prev(1) = 33554444
leon_b_htrans->prev(1) = 2
leon_b_hburst->prev(1) = 0
bus_b_hready->prev(1) = 1

==> 876468 ns: prop4_P4_RTL_CODES_p0.prop4_P4_RTL_CODES_p0 Valid = TRUE <===
==> 876468 ns: prop4_P4_RTL_CODES_p0.prop4_P4_RTL_CODES_p0 Checking = TRUE <===
==> 876468 ns: prop4_P4_RTL_CODES_p0.prop4_P4_RTL_CODES_p0 Pending = FALSE <===

```

```

a_read_addr = 33554432
a_write_length = 33554444
fft_size = 2000
HCLK__rising_CALL = 1
leon_b_wdata = 0
grant_leon_b = 1
leon_b_hwrite = 0
leon_b_adr = 0
leon_b_htrans = 0
leon_b_hburst = 0
bus_b_hready = 1
leon_b_rdata = 0
grant_leon_b->prev(1) = 1
leon_b_hwrite->prev(1) = 0
leon_b_adr->prev(1) = 0
leon_b_htrans->prev(1) = 3
leon_b_hburst->prev(1) = 0
bus_b_hready->prev(1) = 1

```

Selon la première transformation, le moniteur global de l'assertion indique que la trace satisfait la propriété lors de la deuxième programmation du registre `a_read_addr` par le `leon_b` (si une lecture du registre `a_write_length` a été faite et que la valeur lue est égal à `fft_size`). Un exemple de la trace d'exécution est fourni en annexe (D.3.2.1).

Dans le cas de la transformation en SERE, le moniteur global considère que l'assertion est satisfaite sur la trace de simulation quand la valeur de la lecture (`leon_b_HRDATA`) est égal au contenu de la variable `fft_size`. Cette différence est justifiée par la sémantique de satisfaction des opérateurs PSL.

D.3.3 Propriété P_5 raffinée

Le premier paquet est disponible au niveau de l'IO à 0 ns. Ce dernier émet alors une interruption en direction du `leon_a` (`io_module__generate_irq_CALL='1'`). Le `leon_a` configure le `dma_a` qui commence le transfert du paquet vers la mémoire `mem_a` à 168 ns. La fin du transfert du paquet entier a lieu à 54132 ns avant l'arrivée du prochain paquet de données à 700 us. L'assertion est alors satisfaite sur cette trace.

```

-----
IO @ 0 s INFO:  Packet n°0 is now ready to be read
-----

```

```

==> 0 s: rtl_prop5_PROP5_RTL_THESE_p0.rtl_prop5_PROP5_RTL_THESE_p0 Valid    = TRUE <==

```

```

==> 0 s: rtl_prop5_PROP5_RTL_THESE_p0.rtl_prop5_PROP5_RTL_THESE_p0  Checking = FALSE<===
==> 0 s: rtl_prop5_PROP5_RTL_THESE_p0.rtl_prop5_PROP5_RTL_THESE_p0  Pending  = FALSE<===
      io_module__generate_irq_CALL = 1
      io_module_address = 536870912
      HCLK__rising_CALL = 0
      dma_a_HRESP = 0
      dma_a_HGRANT = 0
      dma_a_HWRITE = 0
      dma_a_HTRANS = 0
      dma_a_HBURST = 0
      dma_a_HSIZE = 0
      dma_a_HREADY = 1
      dma_a_HADR = 0
      read_started = 0
.....

```

```

-----
Start transfert to IO module (dma_a_adr == io_module_address)
-----

```

```

==> 168 ns: rtl_prop5_PROP5_RTL_THESE_p0.rtl_prop5_PROP5_RTL_THESE_p0  Valid    = TRUE <===
==> 168 ns: rtl_prop5_PROP5_RTL_THESE_p0.rtl_prop5_PROP5_RTL_THESE_p0  Checking = FALSE<===
==> 168 ns: rtl_prop5_PROP5_RTL_THESE_p0.rtl_prop5_PROP5_RTL_THESE_p0  Pending  = TRUE <===
      io_module__generate_irq_CALL = 0
      io_module_address = 536870912
      HCLK__rising_CALL = 1
      dma_a_HRESP = 0
      dma_a_HGRANT = 1
      dma_a_HWRITE = 0
      dma_a_HTRANS = 2
      dma_a_HBURST = 7
      dma_a_HSIZE = 2
      dma_a_HREADY = 1
      dma_a_HADR = 536870912
      read_started = 1
.....

```

```

-----
End of the transfert to IO module
(dma_a_adr == io_module_address+16000-4)
-----

```

```

====> 54132 ns: rtl_prop5_PROP5_RTL_THESE_p0.rtl_prop5_PROP5_RTL_THESE_p0 Valid      = TRUE <====
====> 54132 ns: rtl_prop5_PROP5_RTL_THESE_p0.rtl_prop5_PROP5_RTL_THESE_p0 Checking = FALSE<====
====> 54132 ns: rtl_prop5_PROP5_RTL_THESE_p0.rtl_prop5_PROP5_RTL_THESE_p0 Pending  = FALSE<====
      io_module__generate_irq_CALL = 0
      io_module_address = 536870912
      HCLK__rising_CALL = 1
      dma_a_HRESP = 0
      dma_a_HGRANT  = 0
      dma_a_HWRITE  = 0
      dma_a_HTRANS  = 3
      dma_a_HBURST  = 7
      dma_a_HSIZE   = 2
      dma_a_HREADY   = 1
      dma_a_HADR    = 536886908
      read_started  = 1

.....

```

```

-----
io_handler_orig_inst @ 700 us INFO:  Packet n°1 is now ready to be read
-----

```

```

====> 700 us: rtl_prop5_PROP5_RTL_THESE_p0.rtl_prop5_PROP5_RTL_THESE_p0 Valid      = TRUE <====
====> 700 us: rtl_prop5_PROP5_RTL_THESE_p0.rtl_prop5_PROP5_RTL_THESE_p0 Checking = TRUE <====
====> 700 us: rtl_prop5_PROP5_RTL_THESE_p0.rtl_prop5_PROP5_RTL_THESE_p0 Pending  = FALSE<====
      io_module__generate_irq_CALL = 1
      io_module_address = 536870912
      HCLK__rising_CALL = 0
      dma_a_HRESP = 0
      dma_a_HGRANT  = 0
      dma_a_HWRITE  = 0
      dma_a_HTRANS  = 0
      dma_a_HBURST  = 0
      dma_a_HSIZE   = 0
      dma_a_HREADY   = 1
      dma_a_HADR    = 0
      read_started  = 0

```

Couche modélisation et transferts en rafale

Pour illustrer l'idée suggérée dans la section 7.2.2 tout en restant au niveau TLM, reprenons l'exemple de la propriété P_3 (c.f. section 6.2.1.1). Nous remarquons que cette assertion utilise une variable auxiliaire “**value**” dont la valeur est calculée dans la couche de modélisation. Dans ce cas, le transfert étant un transfert simple, la valeur de la donnée transférée est calculée en une seule fois, au moment où le composant `leon_a` réalise l'appel à la fonction `write_block()`.

Supposons maintenant qu'il s'agisse d'une suite de rafales au lieu d'un transfert simple. La valeur transférée va devoir être reconstituée à partir des données la constituant. Dans le cas de cette plateforme et de cette assertion, il suffirait de réutiliser dans la couche de modélisation la portion de code itérative déjà utilisée par le concepteur dans son code SystemC pour faire cette même reconstitution, comme indiqué dans la propriété P_{3bis} du programme E.1.

```
vunit prop3bis {  
    // HDL_DECLS  
    const unsigned long ctrl_reg = 16777216 + dma_reg::a_dma_control;  
    const unsigned int start_transfer = dma_reg::dma_control_start_mask;  
    const unsigned int burst_size = 16;  
    const unsigned int transfert_size = burst_size * sizeof(int);  
    unsigned int complete_length, length;  
    unsigned int nb_bursts;  
    unsigned int remaining_words;  
    unsigned char *data_ptr;  
    unsigned int value;  
    unsigned int iterator;  
    // HDL_STMTs :  
    if(leon_a.write_block_CALL())  
    {  
        data_ptr = leon_a.write_block.p2;  
        complete_length = leon_a.write_block.p3;  
        length = complete_length / sizeof(unsigned int);
```



```

    n_bursts = length / burst_size;
    remaining_words = length % burst_size;
    value = 0;
    if(n_bursts){
        for(iterator = 0; iterator < n_bursts; iterator++){
            for(int i = iterator*transfert_size;
                i<iterator*transfert_size+transfert_size;i++)
                value+=data_ptr[i];
        }
    }
    if(remaining_words){
        for(int i= n_bursts*transfert_size;
            i<n_bursts*transfert_size+remaining_words*sizeof(int);i++)
            value+=data_ptr[i];
    }
}

// PROPERTY :
assert always
( (leon_a.write_block_CALL() &&
  leon_a.write_block.p1 == ctrl_reg &&
  value == start_transfer)
  => next(dma_a.generate_irq_CALL()
    before (leon_a.write_block_CALL() &&
      leon_a.write_block.p1 == ctrl_reg &&
      value == start_transfer)
  ));
}

```

PROGRAMME E.1 – Propriété P_3bis au niveau transactionnel

Ici, l'assertion en elle même reste inchangée car le niveau d'abstraction est toujours TLM. Dans le cas où l'assertion devrait de plus être raffinée au niveau RTL, elle ne devrait plus faire référence au début mais à la fin de la transmission, et les instructions de la couche de modélisation permettant de reconstituer la donnée transférée devraient être sensibles à chaque transfert élémentaire.

Bibliographie

- [ABG⁺00] Yael Abarbanel, Ilan Beer, Leonid Glushovsky, Sharon Keidar et Yaron Wolfsthal: *FoCs: Automatic Generation of Simulation Checkers from Formal Specifications*. Dans *Computer Aided Verification (CAV)*, pages 538–542, 2000.
- [ARM08] ARM: *AMBA Specification*. www.arm.com, 2008.
<https://silver.arm.com/download/download.tm?pv=1062782>.
- [Avi10] Ran Avinun: *Validate hardware/software for nextgen mobile/consumer apps using software-on-chip system development tools*. Embedded, December 2010. <http://www.embedded.com/design/mcus-processors-and-socs/4211507/Validate-hardware-software-for-nextgen-mobile-consumer-apps-using-software-on-chip-system-development-tools->.
- [Ayn09] John Aynsley: *The Transaction Level Modeling standard of the Open SystemC Initiative (OSCI)*. Doulo, TLM2.0.1 release, 2007-2009.
- [Bai04] Brian Bailey: *A new vision of scalable verification*. EETimes, Mars 2004. http://www.eetimes.com/document.asp?doc_id=1217606.
- [Bai07] Brian Bailey: *Address verification issues with scalable methods*. EETimes, April 2007. http://www.eetasia.com/ARTICLES/2007APR/PDF/EEOL_2007APR02_TPA_TA.pdf.
- [BCG⁺00] Dhananjay Brahme, Steven Cox, Jim Gallo1, Mark Glasser, William Grundmann, Norris Ip, William Paulsen, John L. Pierce, John Rose, Dean Shea et Karl Whiting: *The Transaction-Based Verification Methodology*. rapport technique, Cadence Berkeley Labs, August 2000. <http://masters.donntu.edu.ua/2007/fvti/smeshkov/library/tbv00tr2.pdf>.
- [BDF08] N. Bombieri, N. Deganello et F. Fummi: *Integrating RTL IPs into TLM Designs Through Automatic Transactor Generation*. Dans *Design, Automation and Test in Europe, 2008. DATE '08*, pages 15–20, March 2008.

- [Ber05] Victor Berman: *A Tale of Two Languages: SystemC and SystemVerilog*. Chip Design Magazine, June 2005. <http://chipdesignmag.com/display.php?articleId=116>.
- [BFF05] N. Bombieri, A. Fedeli et F. Fummi: *On PSL Properties Re-use in SoC Design Flow Based on Transaction Level Modeling*. Dans *Microprocessor Test and Verification, 2005. MTV '05. Sixth International Workshop on*, pages 127–132, Nov 2005.
- [BFG⁺12] N. Bombieri, F. Fummi, V. Guarnieri, G. Pravadelli et S. Vinco: *Redesign and Verification of RTL IPs through RTL-to-TLM Abstraction and TLM Synthesis*. Dans *Microprocessor Test and Verification (MTV), 2012 13th International Workshop on*, pages 76–81, Dec 2012.
- [BFP07] N. Bombieri, F. Fummi et G. Pravadelli: *Incremental ABV for Functional Validation of TL-to-RTL Design Refinement*. Dans *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1–6, April 2007.
- [BGR02] A. Braun, J. Gerlach et W. Rosenstiel: *Checking temporal properties in SystemC specifications*. Dans *High-Level Design Validation and Test Workshop, 2002. Seventh IEEE International*, pages 23–27, Oct 2002.
- [BHAPB14a] Z. Bel Hadj Amor, L. Pierre et D. Borrione: *System-on-Chip Verification :TLM-to-RTL Assertions Transformation*. Dans *PRIME'2014 : 10th Conference on Ph.D. Research in Microelectronics and Electronics, Grenoble, France*, pages 30–3, June-July 2014.
- [BHAPB14b] Z. Bel Hadj Amor, L. Pierre et D. Borrione: *A Tool for the Automatic TLM-to-RTL Conversion of Embedded Systems Requirements for a Seamless Verification Flow*. Dans *VLSI-SoC'2014 : 22nd IFIP/IEEE International Conference on Very Large Scale Integration, Playa del Carmen, Mexico*, pages 6–8, October 2014. To appear.
- [BM10] Brian Bailey et Grant Martin: *ESL Models and their Application*. Embedded Systems. Springer US, 2010.
- [Bou07] Pierre Boumel: *GAUT - High Level Synthesis - From C to RTL*. rapport technique, University of South Britany, March 2007. https://www.rd-access.eu/edatools/system/files/_edaTools/ubooth_submission/2007/GAUT.pdf.
- [BZ05] Marc Boulé et Zeljko Zilic: *Incorporating Efficient Assertion Checkers into Hardware Emulation*. Dans *ICCD '05: Proceedings of the 23rd IEEE International Conference on Computer Design*, pages 221–228, Washington, DC, USA, 2005. IEEE Computer Society.

- [BZ08] Marc Boulé et Zeljko Zilic: *Generating Hardware Assertion Checkers*. Springer, 2008.
- [CAC⁺07] C. Chavet, C. Andriamisaina, P. Coussy, E. Casseau, E. Juin, P. Urard et E. Martin: *A design flow dedicated to multi-mode architectures for DSP applications*. Dans *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*, pages 604–611, Nov 2007.
- [CG03] L. Cai et D. Gajski: *Transaction level modeling: an overview*. Dans *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*, pages 19–24, Oct 2003.
- [CKL04] Edmund Clarke, Daniel Kroening et Flavio Lerda: *A Tool for Checking ANSI-C Programs*. Dans Kurt Jensen et Andreas Podelski (rédacteurs): *Tools and Algorithms for the Construction and Analysis of Systems*, tome 2988 de *Lecture Notes in Computer Science*, pages 168–176. Springer Berlin Heidelberg, 2004.
- [CM04] Koen Claessen et Johan Mårtensson: *An Operational Semantics for Weak PSL*. Dans *Formal Methods in Computer-Aided Design, LNCS 3312*, pages 337–351. Springer-Verlag, 2004.
- [Cor12] Intel Corporation: *Moore's Law: Fun Facts*. Intel, 2012. <http://www.intel.com/content/www/us/en/history/history-moores-law-fun-facts-factsheet.html>.
- [Dou99] Jean Michel Doudoux: *Developpons en JAVA*. Doudoux, Jean-Michel, 1999. <http://www.jmdoudoux.fr/java/dej/chap-serialisation.htm#serialisation>.
- [DR10] Rolf Drechsler et Frank Rogin: *Debugging at the Electronic System Level*. Springer Netherlands, 2010.
- [Duc07] Laurent Ducouso: *Faster Verification is the goal at ST*. EE times europe, Mars 2007. <http://www.nxtbook.com/nxtbooks/cmp/eeteurope031907/index.php?startid=18>.
- [EEH⁺06] W. Ecker, V. Esen, M. Hull, T. Steininger et M. Velten: *Requirements and Concepts for Transaction Level Assertions*. Dans *Computer Design, 2006. ICCD 2006. International Conference on*, pages 286–293, Oct 2006.
- [EESV07] Wolfgang Ecker, Volkan Esen, Thomas Steininger et Michael Velten: *Requirements and Concepts for Transaction Level Assertion Refinement*. Dans Achim Rettberg, Mauro C. Zanella, Rainer Dömer, Andreas Gerstlauer et Franz J. Rammig (rédacteurs): *Embedded System Design: Topics, Techniques and Trends*, tome 231 de *IFIP The International Federation for Information Processing*, pages 1–14. Springer US, 2007, ISBN 978-0-387-72257-3. http://dx.doi.org/10.1007/978-0-387-72258-0_1.

- [Ese08] Volkan Esen: *A New Assertion Language Covering Multiple Levels of Abstraction*. Thèse de doctorat, Technical University Munich, 2008. <http://mediatum2.ub.tum.de/doc/644594/document.pdf>.
- [Fer11] Luca Ferro: *Vérification de propriétés logico-temporelles de spécifications SystemC TLM*. Thèse de doctorat, Université de Grenoble, 2011. <http://www.theses.fr/2011GRENT032/document>.
- [FKL04] Harry Foster, Adam Krolnik et David Lacey: *Assertion-Based Design 2nd Edition*. Information Technology: Transmission, Processing and Storage. Kluwer Academic, May 2004.
- [Fos09] Harry Foster: *Applied Assertion-Based Verification: An Industry Perspective*. Foundations and Trends in Electronic Design Automation, 3(1):1–95, 2009.
- [Fos11] Harry D. Foster: *Mentor Graphics Challenges of Design and Verification in the SoC Era Trends, Predictions, and Forecast*, Nov 2011. http://testandverification.com/files/DVConference2011/2_Harry_Foster.pdf.
- [Fos13] Harry Foster: *Part 5: The 2012 Wilson Research Group Functional Verification Study*. Mentor Graphics, July 2013. <http://blogs.mentor.com/verificationhorizons/blog/2013/07/15/part-5-the-2012-wilson-research-group-functional-verification-study/>.
- [FP09] L. Ferro et L. Pierre: *ISIS: Runtime verification of TLM platforms*. Dans *Specification Design Languages, 2009. FDL 2009. Forum on*, pages 1–6, Sept 2009.
- [Fuj03] Masahiro Fujita: *System level design methodologies from the viewpoint of formal verification*. Dans *ASIC, 2003. Proceedings. 5th International Conference on*, tome 1, pages 6–10 Vol.1, 2003.
- [GCMC⁺05] Frank Ghenassia, Alain Clouard, Laurent Maillet-Contoz, Jean Philippe Strassen, Eric Paire, Thibaut Bultiaux, Stephane Guenot, Serge Hustin, Alexandre Blampey, Joseph Bulone, Matthieu Moy, Antoine Perrin, Gregory Poivre, Christophe Amerijckx et Amine Kerkeni: *Transaction-Level Modeling with SystemC : TLM Concepts and Applications for Embedded Systems*. Springer, 2005.
- [GD03] Daniel Große et Rolf Drechsler: *Formal Verification of LTL Formulas for SystemC Designs*. Dans *ISCAS '03: Proceedings of the 2003 International Symposium on Circuits and Systems*, tome 5, pages 245–248, May 2003.
- [Ger10] Anderas Gerstlauer: *System Design Methodology*. university of texas, 2010. <http://www.cerc.utexas.edu/jaa/soc/lectures.php?lecture=8-2.pdf>.

- [GGP08] Patrice Gerin, Xavier Guérin et Frédéric Pétrot: *Efficient Implementation of Native Software Simulation for MPSoC*. Dans *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 676–681, 2008. <http://doi.acm.org/10.1145/1403375.1403540>.
- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, October 1994.
- [GLD10] D. Große, H.M. Le et R. Drechsler: *Proving transaction and system-level properties of untimed SystemC TLM designs*. Dans *Formal Methods and Models for Codesign (MEMOCODE)*, 2010 8th IEEE/ACM International Conference on, pages 113–122, July 2010.
- [GLMS02] Thorsten Grotker, Stan Liao, Grant Martin et Stuart Swan: *System Design with Systemc*. Morgan Kaufmann, 2002.
- [HFK⁺07] Christian Haubelt, Joachim Falk, Joachim Keinert, Thomas Schlichter, Martin Streubühr, Andreas Deyhle, Andreas Hadert et Jürgen Teich: *A SystemC-Based Design Methodology for Digital Signal Processing Systems*. EURASIP Journal on Embedded Systems, 2007(1):047580, 2007. <http://jes.eurasipjournals.com/content/2007/1/047580>.
- [HR04] Michael Huth et Mark Ryan: *Logic in Computer Science :modelling and reasoning about systems*. Cambridge University Press, June 2004.
- [HT05] Ali Habibi et Sofiene Tahar: *Design for Verification of SystemC Transaction Level Models*. Dans *DATE '05: Proceedings of the 2005 conference on Design, Automation and Test in Europe*, pages 560–565, 2005.
- [HT06] Ali Habibi et Sofiene Tahar: *Design and Verification of SystemC Transaction-Level Models*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 14(1):57–68, January 2006.
- [KB06] Tim Kogel et Matthew Braun: *Virtual Prototyping of Embedded Platforms for Wireless and Multimedia*. Dans *DATE '06: Proceedings of the Conference on Design, Automation and Test in Europe, Munich, Germany*, tome 1, pages 1–6, March 2006.
- [KBS11] Rishabh Singh Kurmi, Shruti Bhargava et Ajay Somkuw: *Design of AHB Protocol Block for Advanced Microcontrollers*. International Journal of Computer Applications, 32(8), October 2011. <http://research.ijcaonline.org/volume32/number8/pxc3875543.pdf>.
- [Kea11] Michael Keating: *The simple art of SoC design*. Springer, Dordrecht, 2011.
- [KT07] Atsushi Kasuya et Tesh Tesfaye: *Verification Methodologies in a TLM-to-RTL Design Flow*. Dans *Proceedings of the 44th Annual Design Automation*

- Conference*, DAC '07, pages 199–204, New York, NY, USA, 2007. ACM. <http://doi.acm.org/10.1145/1278480.1278529>.
- [Kul08] Nihal Kularatna: *Electronic Circuit Design: From Concept to Implementation*. CRC Press, June 2008.
- [Lah06] Younes Lahbib: *Extension of Assertion-Based Verification Approaches for the Verification of SystemC SoC models*. Thèse de doctorat, Faculty of Science of Monastir, Monastir, Tunisia, December 2006.
- [LBC⁺10] Vincent Lefftz, Jean Bertrand, Hugues Casse, Christophe Clienti, Philippe Coussy, Laurent Maillet-Contoz, Philippe Mercier, Pierre Moreau, Laurence Pierre et Emmanuel Vaumorin: *A design flow for critical embedded systems*. Dans *Industrial Embedded Systems (SIES), 2010 International Symposium on*, pages 229–233, 2010.
- [Lez13] Gabriel Lezmi: *EDA tools and IP are still design differentiators, says Synopsys*. ElectronicsWeekly.com, December 2013. <http://www.electronicweekly.com/news/business/viewpoints/eda-tools-and-ip-are-still-design-differentiators-says-synopsys-2013-12/>.
- [MAB06] Katell Morin-Allory et Dominique Borrione: *Proven correct monitors from PSL specifications*. Dans *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, tome 1, pages 1–6, March 2006.
- [MAGB07] K. Morin-Allory, E. Gascard et D. Borrione: *Synthesis of Property Monitors for Online Fault Detection*. Journal of Circuits, Systems and Computers, 16(06):943–960, 2007. <http://www.worldscientific.com/doi/abs/10.1142/S0218126607004088>.
- [Man13] Tets Maniwa: *Focus Report: Electronic System-Level (ESL) Tools*. issue of Chip Design Magazine, 2013. <http://chipdesignmag.com/display.php?articleId=23>.
- [MAOB08] Katell Morin-Allory, Yann Oddos et Dominique Borrione: *Horus: A tool for Assertion-Based Verification and on-line testing*. Dans *MEMOCODE '08: In Proceeding of the 6th ACM/IEEE International Conference on Formal Methods and Models for Codesign*, June 2008.
- [Mar03] Grant Martin: *SystemC and the future of design languages: opportunities for users and research*. Dans *SBCCI 2003 : Proceedings 16th Symposium on Integrated Circuits and Systems Design, 2003.*, pages 61–62, 2003.
- [MBP07] Grant Martin, Brian Bailey et Andrew Piziali: *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann Publishers Inc., 2007, ISBN 9780080488837.

- [Men06] MentorGraphics: *Verification Cookbook :Advanced Verification Methodology*. MentorGraphics, March 2006. Version 1.3.
- [NH06] Bernhard Niemann et Christian Haubelt: *Assertion-Based Verification of Transaction Level Models*. Dans Bernd Straube et Martin Freibothe (rédacteurs): *MBMV*, pages 232–236. Fraunhofer Institut für Integrierte Schaltungen, 2006.
- [OMAB08] Y. Oddos, K. Morin-Allory et D. Borriore: *Assertion-Based Design with Horus*. Dans *Formal Methods and Models for Co-Design, 2008. MEMOCODE 2008. 6th ACM/IEEE International Conference on*, pages 75–76, June 2008.
- [OMG09] OMG: *OMG Unified Modeling Language (OMG UML), Superstructure*, February 2009. <http://www.omg.org/spec/UML/2.2>, Version 2.2.
- [ope10] opencores: *WISHBONE, Revision B.4 Specification*. openCores.org, 2010. http://cdn.opencores.org/downloads/wbspec_b4.pdf.
- [OSC05] OSCI: *OSCI TLM 1.0 Language Reference Manual*, April 2005. <http://www.accellera.org/downloads/standards/systemc>, version 1.0.
- [OSC09] OSCI: *OSCI TLM 2.0 Language Reference Manual*, July 2009. <http://www.accellera.org/downloads/standards/systemc>, version 2.0.1.
- [PBHA13] L. Pierre et Z. Bel Hadj Amor: *Automatic refinement of requirements for verification throughout the SoC design flow*. Dans *Hardware/Software Code-sign and System Synthesis (CODES+ISSS), 2013 International Conference on*, pages 1–10, Sept 2013.
- [PF10] L. Pierre et L. Ferro: *Enhancing the assertion-based verification of TLM designs with reentrancy*. Dans *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 103–112, July 2010.
- [PSL05] *IEEE Std 1850-2005, IEEE Standard for Property Specification Language (PSL)*, October 2005.
- [PSL10] *IEEE Std 1850-2010, IEEE Standard for Property Specification Language (PSL),(Revision of IEEE Std1850-2005) - Redline*, April 2010. <http://standards.ieee.org/findstds/standard/1850-2010.html>.
- [RHKR01] J. Ruf, D.W. Hoffmann, T. Kropf et W. Rosenstiel: *Simulation-guided property checking based on multi-valued AR-automata*. Dans *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, pages 742–748, 2001.
- [RN03] Ben Rodes et Dan Notestein: *Coding techniques for Bus Functional Models In Verilog, VHDL and C++*.

- rapport technique, SynaptiCAD, September 2003.
http://www.syncad.com/paper_coding_techniques_icu_sep_2003.htm.
- [RSPF05] Adam Rose, Stuart Swan, John Pierce et Jean Michel Fernandez: *tlm1 whitepaper: Transaction Level Modeling in SystemC*. rapport technique, Cadence Design Systems, August 2005.
http://www.it.kth.se/courses/IL2452/Aug2012/Lectures/tlm_whitepaper.pdf.
- [RSV97] James A. Rowson et Alberto Sangiovanni-Vincentelli: *Interface-based Design*. Dans *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, pages 178–183, New York, NY, USA, 1997. ACM, ISBN 0-89791-920-3. <http://doi.acm.org/10.1145/266021.266060>.
- [SC005] *IEEE Std 1666-2005, IEEE Standard System C Language Reference Manual*, March 2005.
- [Sch09] Frank Schirrmeister: *TLM-2.0 APIs Open SystemC To Mainstream Virtual Platform Adoption*. Electronic Design, April 2009.
<http://electronicdesign.com/products/tlm-20-apis-open-systemc-mainstream-virtual-platform-adoption>.
- [Sch13] Philip Schulz: *wb_switch - A Wishbone Bus Interconnect for FPGAs*, January 2013. http://www.phisch.org/website/wb_switch/.
- [SCV06] *OSCI, SystemC Verification Library*, June 2006.
- [SO14] Pravin S. Shete et Shruti Oza: *Design of an Efficient FSM for an Implementation of AMBA AHB Master*. International Journal of Advanced Research in Computer Science and Software Engineering, 4(3), March 2014.
http://www.ijarcse.com/docs/papers/Volume_4/3_March2014/V4I3-0325.pdf.
- [Ste12] Thomas Steininger: *Automated assertion transformation across multiple abstraction levels*. Thèse de doctorat, Technical University Munich, 2012.
<http://mediatum.ub.tum.de/doc/676680/676680.pdf>.
- [SV005] *IEEE Std 1800-2005, IEEE Standard for System Verilog - Unified Hardware Design, Specification and Verification Language*, 2005.
- [SV009] *IEEE Std 1800-2009, IEEE Standard for System Verilog - Unified Hardware Design, Specification, and Verification Language*, December 2009.
- [Swa06] S. Swan: *SystemC transaction level models and RTL verification*. Dans *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 90–92, 2006.
- [TS08] B. Turumella et M. Sharma: *Assertion-based verification of a 32 thread SPARC x2122; CMT microprocessor*. Dans *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 256–261, June 2008.

- [VER01] *IEEE Std 1364-2001, IEEE Standard Verilog Hardware Description Language*, September 2001.
- [VHD02] *IEEE Std 1076-2002, IEEE Standard VHDL Language Reference Manual*, 2002.
- [VV005] *IEEE Std 1012-2004 (Revision of IEEE Std 1012-1998), IEEE Standard for Software Verification and Validation*, 2005.
- [Wal05] Robert Dale Walstrom: *System-level design refinement using SystemC*. Thèse de doctorat, Iowa State University, 2005.
- [WGR05] Bruce WILE, John C. GOSS et Wolfgang ROESNER: *Comprehensive Functional Verification*. Kaufmann, Morgan, 2005.

Glossaire

A

- ABV *Assertion Based Verification*
ABD *Assertion Based Design*
AT *Approximatly Timed*
ASIC *Application Specific Integrated Circuit*
ASM *Abstract State Machine*
ASML *Abstract State Machine Language*

B

C

- CA *Cycle Accurate*
CI *Integrated Circuit*
CPU *Central Processing Unit*

D

- DMA *Direct Memory Access*
DSP *Digital Signal Processor*
DTD *Document Type Definition*
DUV *Design Under Verification*

E

- EFSM *Extended Finite State Machine*
ESL *Electronic Sustem Level*
EBS *Event Based Simultaneity*

F

- FA *Functionally Accurate*
FSM *Finite State Machine*
FVTC *Formal Verification Technical Commitee*

FLTL *Finite Linear time Temporal Logic*

G

GDL *General Description Language*

H

HDL *Hardware Description Language*

I

IMC *Interface Method Call*

iOS *integrated Operating System*

IP *silicon Intellectual Property*

ISA *Instruction Set Architecture*

J

K

kIPS *thousand Instructions Per Second*

L

LRM *Language Reference Manual*

LT *Loosely Timed*

M

MIPS *Million Instructions Per Second*

MPSoC *Multi-Processor System-on-Chip*

N

NoC *Network-on-Chip*

O

OSCI *Open SystemC Initiative*

OCP *Open Core Protocol*

P

PA *Protocol Accurate*

PSL *Property Specification Language*

PV *Programmer View*

R

RTL *Register Transfer Level*

S

SERE *Sequential Extended Regular Expressions*

SL *System Level*

SLDL *System Level Design Language*

SoC *System-on-Chip*

T

TA *Transaction Accurate*

TIMA *Techniques de l'Informatique et de la Microélectronique pour l'Architecture des systèmes intégrés*

TLM *Transaction Level Modeling*

TTM *Time To Market*

U

V

VDC *Value Change Dump*

Vhic *Very High Speed Integrated Circuit*

VHDL *Vhic Hardware Description Language*

VLSI *Very Large Scale Integration*

X

XML *eXtensible Markup Language*

XSD *XML Schema Description*

Index

- événements, 30, 31
- checker*, 13
- abstraction, 7
- ABV, 13–16, 52, 67–69, 75
- Application View, 7
- Approximatly timed, 45
- assertion, 13
- bit-accurate, 6, 25, 26, 48, 86
- canaux, 27
- CTL, 52, 53, 55
- Cycle Approximative, 7
- cycle-accurate, 5, 26, 31, 45, 67, 69, 90, 92
- DMA, 38, 39, 77, 78, 80–83
- ESL, 4–7, 9, 10, 12, 16, 22, 26, 43, 46, 48, 50
- FL, 53, 54, 56–60, 64–66, 77, 97, 98, 101, 127, 129, 130, 132, 133, 162, 168, 213, 214
- flot classique, 2
- generic payload, 34
- HDL, 2
- Horus, 14, 16, 65–67, 75, 76, 83, 104, 109, 124–127, 129, 130, 172
- IMC, 29, 34
- interfaces, 28
- interfaces de transport, 35
- ISIS, 16, 19, 67, 73, 75–80, 83, 100, 104, 105, 108, 109, 111, 121–124, 126, 127, 129, 130, 132–134, 159, 172, 188, 193
- kernel, 27, 31
- Loosely timed, 45
- LTL, 52–55, 60, 68
- Module, 27
- moniteur, 13
- niveaux d’abstraction, 46
- pin-accurate, 25, 26, 41, 48, 86, 95
- ports, 29
- processus, 30
- Programmer’s View, 7
- PSL, 14, 18, 52–59, 63–71, 73–81, 95–100, 102, 108, 109, 111, 116–119, 122, 124, 137, 139, 141, 142, 147, 161, 168, 169, 171, 172, 174, 176, 246
- PSL_{ss}, 56, 60–62, 65, 77, 96, 97, 99, 104, 132
- pv_dma, 37, 90
- raffinement, 9, 16, 44, 48, 49, 83, 91, 95
- RTL, 2–4, 6, 9–12, 14, 16–18, 22–27, 37, 41, 43, 45, 46, 48–50, 52, 58, 64, 65, 67, 76, 81–83, 86–88, 90, 92–94, 96, 104, 105, 108, 109, 113, 116, 122, 134, 136, 140, 142, 143, 153–155, 157, 161,

163–165, 167, 172–174, 186, 188, 226,
246

SERE, 55, 56, 61–63, 76, 79, 81, 82, 100, 104,
105, 108, 109, 124–133, 159, 162, 168,
172, 176, 213, 214, 222

simulation, 12

SystemC, 10, 11, 26–33, 39, 41, 45, 53, 67–
69, 74–76

TLM, 9–12, 15–18, 22, 26, 32–39, 41, 43–47,
49, 50, 52, 67, 69, 70, 75, 76, 78, 83,
86, 87, 92, 96, 105, 108, 109, 122,
136, 144, 164, 167, 171, 174, 181,
197, 225, 226, 246

transport bloquant, 35

transport non bloquant, 36

vérification, 12

vérification dynamique, 12

vérification statique, 13

VHDL, 2, 31, 53, 59, 66, 67, 76

wishbone, 23, 49

Publications de l'Auteur

Conférences internationales avec comité de lecture, Sélection sur le papier complet

- L. Ferro, L. Pierre, Z. Bel Hadj Amor, J. Lachaize, V. Lefftz, “*Runtime Verification of Typical Requirements for a Space Critical SoC Platform*”, Proceeding of the 16th International Workshop on Formal Methods for Industrial Critical Systems FMICS’11, Trento (Italy), August 29-30, 2011. Publisher : Springer, Pages : Volume 6959/201, pp. 21-3.
- L. Pierre, L. Ferro, Z. Bel Hadj Amor, P. Bourgon , J. Quévremont, “*Integrating PSL Properties into SystemC Transactional Modeling - Application to the Verification of a Modern SoC*”, Proceeding of the IEEE International Symposium on Industrial Embedded Systems (SIES’2012), Karlsruhe (Germany), June 20-22, 2012.
- L. Pierre, Z. Bel Hadj Amor, “*Automatic Refinement of Requirements for Verification throughout the SoC Design Flow*”, Proceeding of the International Conference on Hardware/Software Codesign and System Synthesis CODES+ISSS’13, Embedded System Week, Montreal (Canada), September 29 - October 4, 2013.
- Z. Bel Hadj Amor, L. Pierre, D. Borriane “*System-on-Chip Verification : TLM-to-RTL Assertions Transformation*”, Proceeding of the 10th Conference on Ph.D. Research in Microelectronics and Electronics PRIME’14, Grenoble (France), June 30, July 3, 2014.
- Z. Bel Hadj Amor, L. Pierre, D. Borriane, “*A Tool for the Automatic TLM-to-RTL Conversion of Embedded Systems Requirements for a Seamless Verification Flow*”, Proceeding of 22th IFIP/IEEE International Conference on Very Large Scale Integration VLSI-SoC’14, Playa del Carmen (Mexico), October 6-8, 2014.

Résumé – Mots Clefs : ABV, vérification, raffinement, SystemC, TLM, RTL, PSL. Cette thèse se situe dans le contexte de la vérification fonctionnelle des circuits intégrés complexes. L'objectif de ce travail est de créer un flot de vérification conjoint au flot de conception basé sur une technique appelée "vérification basée sur les assertions (ABV)". Le concept de base du flot est le raffinement automatique des spécifications formelles données sous la forme d'assertions PSL du niveau TLM au niveau RTL. La principale difficulté est la disparité des deux domaines : au niveau TLM, les communications sont modélisées par des appels de fonctions atomiques. Au niveau RTL, les échanges sont assurés par des signaux binaires évoluant selon un protocole de communication précis. Sur la base d'un ensemble de règles de transformation temporelles formelles, nous avons réalisé un outil permettant d'automatiser le raffinement de ces spécifications. Comme le raffinement des modèles, le raffinement des assertions n'est pas entièrement automatisable : des informations temporelles et structurelles doivent être fournies par l'utilisateur. L'outil réalise la saisie de ces informations de façon ergonomique, puis procède automatiquement à la transformation temporelle et structurelle de l'assertion. Il permet la génération d'assertions RTL mais aussi hybrides. Les travaux antérieurs dans ce domaine sont peu nombreux et les solutions proposées imposent de fortes restrictions sur les assertions considérées. À notre connaissance, le prototype que nous avons mis en œuvre est le premier outil qui réalise un raffinement temporel fondé sur la sémantique formelle d'un langage de spécification standard (PSL).

Abstract – Key words : ABV, verification, refinement, SystemC, TLM, RTL, PSL. The context of this thesis is the functional verification of complex integrated circuits. The objective of our work is to create a seamless verification flow joint to the design flow and based on a proved technique called Assertions-Based Verification (ABV). The main challenge of TLM to RTL refinement is the disparity of these two domains : at TLM, communications are modeled as atomic function calls handling all the exchanged data. At RTL, communications are performed by signals according to a specific communication protocol. The proposed temporal transformation process is based on a set of formal transformation rules. We have developed a tool performing the automatic refinement of PSL specifications. As for design refinement assertion refinement is not fully automated. Temporal and structural information must be provided by the user, using an ergonomic interface. The tool allows the generation of assertions in RTL but also hybrid assertions. Little work has been done before in this area, and the proposed solutions suffer from severe restrictions. To our knowledge, our prototype is the first tool that performs a temporal transformation of assertions based on the formal semantics of a standard specification language (PSL).

Adresse : Laboratoire TIMA, 46 Avenue Félix Viallet, 38031 Grenoble Cedex, France.

ISBN : 978-2-11-129196-6